Pandas 1

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

Introduction to Pandas

Pandas Installation

In [3]:
```
# !pip install pandas    -- remove hashtag and run this command if pandas is
```

Importing Pandas

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

In [4]:
```
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.

Now how should we read this dataset?

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

In [5]:
```
df = pd.read_csv('mckinsey.csv') # storing the data in df
df
```

Out[5]:

	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 [6]:
```
type(df)
```

Out[6]:
```
pandas.core.frame.DataFrame
```

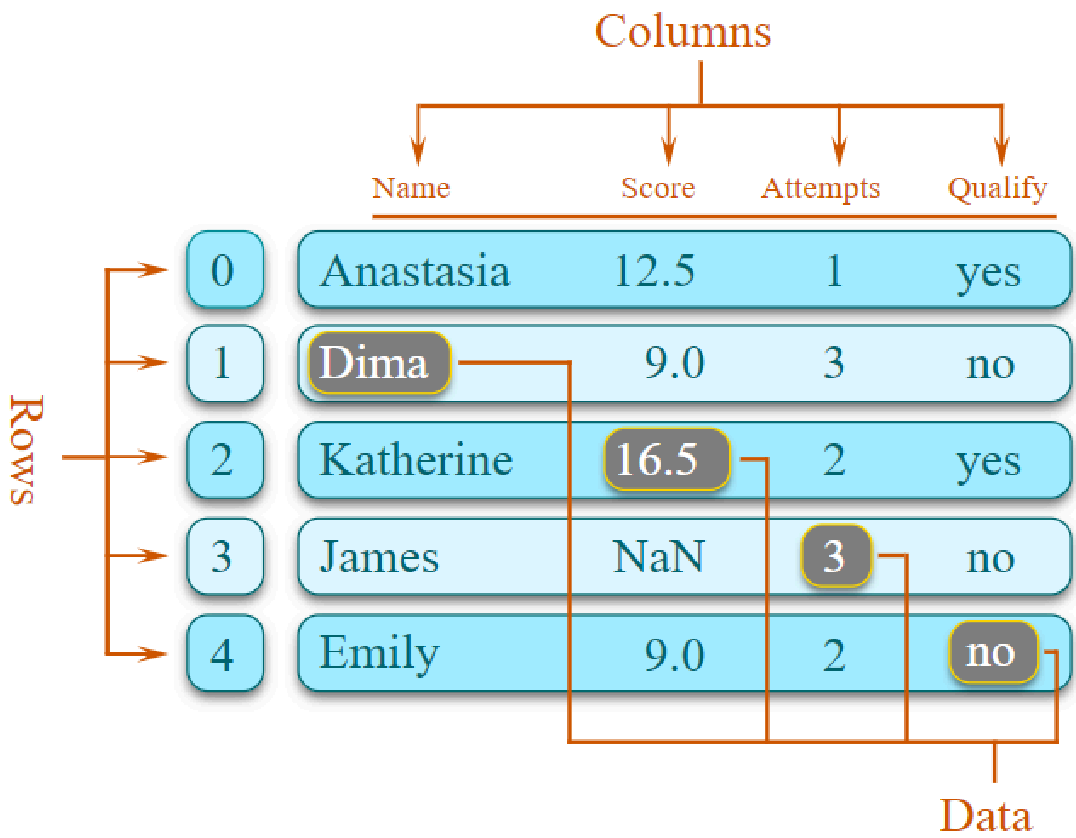
It is a **Pandas DataFrame**

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.

In [7]:
```
from IPython.display import Image
Image(filename='download.png')
```

Out[7]:



How can we access a column, say `country` of the dataframe?

In [8]:
```
df["country"]
```

Out[8]:
```
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 present in the **country** column.

What is the data-type of a column?

In [9]:
```
type(df["country"])
```

Out[9]:
```
pandas.core.series.Series
```

It is a **Pandas Series**

What is a Pandas Series?

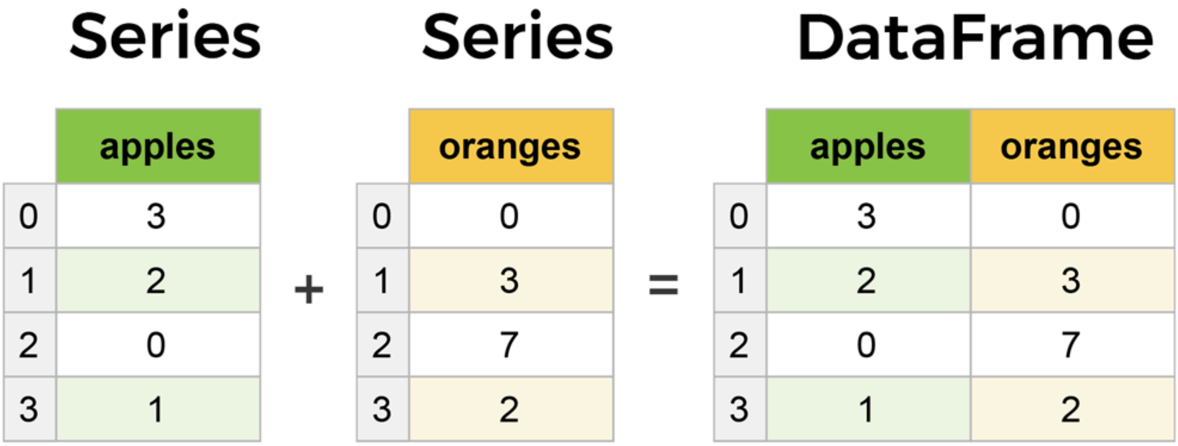
- A **Series** in Pandas is what a **Vector** is in Numpy.

What exactly does that mean?

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

In [10]:
```
from IPython.display import Image
Image(filename='series.png')
```

Out[10]:



Now we have understood what Series and DataFrame are.

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

In [11]:
```
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** 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**.

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

In [12]:
```
df.head()
```

Out[12]:

	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

`df.head()` prints the top 5 rows by default.

We can also pass in number of rows that we want to see.

In [13]:
```
df.head(10)
```

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
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

Similarly, we can use **`df.tail()` if we wish to see the last few rows**.

In [14]:
```
df.tail()
```

Out[14]:

	country	year	population	continent	life_exp	gdp_cap
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 a dataframe?

In [15]:
```
df.shape
```

Out[15]:
```
(1704, 6)
```

Similar to Numpy, it gives the **no. of rows and columns**.

Basic operations on Columns

What operations can we do using columns?

- Add a column
- Delete a column
- Rename a column

We can see that our dataset has 6 columns.

How can we get the names of all these cols?

We can do it in two ways:

1. `df.columns`
2. `df.keys`

In [16]:
```
df.columns  # using attribute `columns` of dataframe
```

Out[16]:
```
Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_ca
p'], dtype='object')
```

In [17]:
```
df.keys()  # using method `keys()` of dataframe
```

Out[17]:
```
Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_ca
p'], 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.

How can we access these columns?

In [18]:
```
df['country'].head()  # accessing a single column
```

Out[18]:
```
0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object
```

In [19]:
```
df[['country', 'life_exp']].head() # accessing multiple columns
```

Out[19]:

	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?

In [20]:
```
df[['country']].head()
```

Out[20]:

	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 a Series while `[['country']]` gives a DataFrame.

How can we find the countries that have been surveyed?

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

In [21]:
```
df['country'].unique()
```

Out[21]:
```
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)
```

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

In [22]:
```
df['country'].value_counts()
```

Out[22]:
```
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`

In [23]:
```
df.rename({"population": "Population", "country":"Country" }, axis = 1)
```

Out[23]:

	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 specifying `axis`
- by using the `column` parameter

In [24]:
```
df.rename(columns={"country":"Country"})
```

Out[24]:

	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` -

In [25]:
```
df
```

Out[25]:

	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.

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

In [26]:

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

Out[26]:

	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.

There's another way of accessing the column values.

In [27]:
```
df.Country
```

Out[27]:
```
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 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`

We already know the continents in which each country lies.

So we probably don't need this column.

How can we delete columns from a dataframe?

In [28]:
```
df.drop('continent', axis=1)
```

Out[28]:

	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:

- 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()`.

In [29]:
```
df.drop(columns=['continent'])
```

Out[29]:

	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, the column `contintent` is dropped.

Has the column been permanently deleted?

In [30]:
```
df.head()
```

Out[30]:

	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 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`.

In [31]:
```
df.drop('continent', axis=1, inplace=True)
```

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?

In [32]:
```
df["year+7"] = df["year"] + 7
df.head()
```

Out[32]:

	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`?

In [33]:
```
df['gdp'] = df['gdp_cap'] * df['population']
df.head()
```

Out[33]:

	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` columns**.

What other operations we can use?

- Addition
- Subtraction
- Division

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.

In [34]:
```
df["Own"] = [i for i in range(1704)]  # count of these values should be corr
df
```

Out[34]:

	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

Before we move to ops on rows, let's drop the newly created columns.

In [35]:
```
df.drop(columns=["Own",'gdp', 'year+7'], axis = 1, inplace = True)
df
```

Out[35]:

	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

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)?**

In [36]:

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

Out[36]:

	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 now starts from 1 instead of 0.

Explicit & 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**.

In [37]:
```
df.index[1] # implicit index 1 gave explicit index 2
```

Out[37]: 2

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`

In [38]:
```
df.index = np.arange(1, df.shape[0]+1, dtype='float')
df
```

Out[38]:

	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 now floating point values.

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

In [39]:
```
sample = df.head()
sample
```

Out[39]:

	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

What if we want to use string indices?

In [40]:
```
sample.index = ['a', 'b', 'c', 'd', 'e']
sample
```

Out[40]:

	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 that we can use almost anything as our explicit index.

Now, let's reset our indices back to integers.

In [41]:
```
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.

In [42]:
```
ser = df["Country"]
ser.head(20)
```

Out[42]:
```
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 13th element?

In [43]:
```
ser[12]
```

Out[43]:
```
'Afghanistan'
```

What about accessing a subset of rows (say 6th to 15th)?

In [44]:
```
ser[5:15]
```

Out[44]:
```
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.

How can we access a row in a dataframe?

In [46]:
```
df[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:3
802, in Index.get_loc(self, key, method, tolerance)
   3801 try:
-> 3802     return self._engine.get_loc(casted_key)
   3803 except KeyError as err:

File ~/anaconda3/lib/python3.11/site-packages/pandas/_libs/index.pyx:138, i
n pandas._libs.index.IndexEngine.get_loc()

File ~/anaconda3/lib/python3.11/site-packages/pandas/_libs/index.pyx:165, i
n pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:5745, in pandas._libs.hashtabl
e.PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:5753, in pandas._libs.hashtabl
e.PyObjectHashTable.get_item()

KeyError: 0

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

KeyError                                  Traceback (most recent call last)
Cell In[46], line 1
----> 1 df[0]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/frame.py:3807, in
DataFrame.__getitem__(self, key)
   3805 if self.columns.nlevels > 1:
   3806     return self._getitem_multilevel(key)
-> 3807 indexer = self.columns.get_loc(key)
   3808 if is_integer(indexer):
   3809     indexer = [indexer]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:3
804, in Index.get_loc(self, key, method, tolerance)
   3802     return self._engine.get_loc(casted_key)
   3803 except KeyError as err:
-> 3804     raise KeyError(key) from err
   3805 except TypeError:
   3806     # If we have a listlike key, _check_indexing_error will raise
   3807     #  InvalidIndexError. Otherwise we fall through and re-raise
   3808     #  the TypeError.
   3809     self._check_indexing_error(key)

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?

In [47]:
```
df[5:15]
```

Out[47]:

	Country	year	population	life_exp	gdp_cap
6	Afghanistan	1977	14880372	38.438	786.113360
7	Afghanistan	1982	12881816	39.854	978.011439
8	Afghanistan	1987	13867957	40.822	852.395945
9	Afghanistan	1992	16317921	41.674	649.341395
10	Afghanistan	1997	22227415	41.763	635.341351
11	Afghanistan	2002	25268405	42.129	726.734055
12	Afghanistan	2007	31889923	43.828	974.580338
13	Albania	1952	1282697	55.230	1601.056136
14	Albania	1957	1476505	59.280	1942.284244
15	Albania	1962	1728137	64.820	2312.888958

Woah, so the slicing works.

This can be a cause for confusion.

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

loc and iloc

1. loc

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

In [51]:
```
df.loc[1]
```

Out[51]:
```
Country        Afghanistan
year                  1952
population         8425333
life_exp            28.801
gdp_cap         779.445314
Name: 1, dtype: object
```

In [52]:
```
df.loc[1:3]
```

Out[52]:

	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 index.

In [53]:
```
df.iloc[1]
```

Out[53]:
```
Country       Afghanistan
year                 1957
population        9240934
life_exp           30.332
gdp_cap         820.85303
Name: 2, dtype: object
```

Will `iloc` also consider the range inclusive?

In [54]:
```
df.iloc[0:2]
```

Out[54]:

	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.

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

In [55]:
```
df.iloc[[1, 10, 100]]
```

Out[55]:

	Country	year	population	life_exp	gdp_cap
2	Afghanistan	1957	9240934	30.332	820.853030
11	Afghanistan	2002	25268405	42.129	726.734055
101	Bangladesh	1972	70759295	45.252	630.233627

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 [56]:
```
df.iloc[-1]

# Works and gives last row in dataframe
```

Out[56]:
```
Country         Zimbabwe
year                2007
population      12311143
life_exp          43.487
gdp_cap       469.709298
Name: 1704, dtype: object
```

In [57]:
```
df.loc[-1]

# Does not work
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:3
802, in Index.get_loc(self, key, method, tolerance)
   3801 try:
-> 3802     return self._engine.get_loc(casted_key)
   3803 except KeyError as err:

File ~/anaconda3/lib/python3.11/site-packages/pandas/_libs/index.pyx:138, i
n pandas._libs.index.IndexEngine.get_loc()

File ~/anaconda3/lib/python3.11/site-packages/pandas/_libs/index.pyx:165, i
n pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:2263, in pandas._libs.hashtabl
e.Int64HashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:2273, in pandas._libs.hashtabl
e.Int64HashTable.get_item()

KeyError: -1

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

KeyError                                  Traceback (most recent call last)
Cell In[57], line 1
----> 1 df.loc[-1]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1073,
in _LocationIndexer.__getitem__(self, key)
   1070 axis = self.axis or 0
   1072 maybe_callable = com.apply_if_callable(key, self.obj)
-> 1073 return self._getitem_axis(maybe_callable, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1312,
in _LocIndexer._getitem_axis(self, key, axis)
   1310 # fall thru to straight lookup
   1311 self._validate_key(key, axis)
-> 1312 return self._get_label(key, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1260,
in _LocIndexer._get_label(self, label, axis)
   1258 def _get_label(self, label, axis: int):
   1259     # GH#5567 this will fail if the label is not present in the axi
s.
-> 1260     return self.obj.xs(label, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/generic.py:4056,
in NDFrame.xs(self, key, axis, level, drop_level)
   4054             new_index = index[loc]
   4055 else:
-> 4056     loc = index.get_loc(key)
   4058     if isinstance(loc, np.ndarray):
   4059         if loc.dtype == np.bool_:

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:3
804, in Index.get_loc(self, key, method, tolerance)
   3802     return self._engine.get_loc(casted_key)
   3803 except KeyError as err:
-> 3804     raise KeyError(key) from err
   3805 except TypeError:
   3806     # If we have a listlike key, _check_indexing_error will raise
   3807     #  InvalidIndexError. Otherwise we fall through and re-raise
   3808     #  the TypeError.
```

```
   3809     self._check_indexing_error(key)

KeyError: -1
```

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 [58]:
```
temp = df.set_index("Country")
temp
```

Out[58]:

Country	year	population	life_exp	gdp_cap
Afghanistan	1952	8425333	28.801	779.445314
Afghanistan	1957	9240934	30.332	820.853030
Afghanistan	1962	10267083	31.997	853.100710
Afghanistan	1967	11537966	34.020	836.197138
Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...
Zimbabwe	1987	9216418	62.351	706.157306
Zimbabwe	1992	10704340	60.377	693.420786
Zimbabwe	1997	11404948	46.809	792.449960
Zimbabwe	2002	11926563	39.989	672.038623
Zimbabwe	2007	12311143	43.487	469.709298

1704 rows × 4 columns

Note: In earlier versions of Pandas, `drop=True` has to be provided to delete the column being used as new index.

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

In [59]:
```
temp.loc['Afghanistan']
```

Out[59]:

	year	population	life_exp	gdp_cap
Country				
Afghanistan	1952	8425333	28.801	779.445314
Afghanistan	1957	9240934	30.332	820.853030
Afghanistan	1962	10267083	31.997	853.100710
Afghanistan	1967	11537966	34.020	836.197138
Afghanistan	1972	13079460	36.088	739.981106
Afghanistan	1977	14880372	38.438	786.113360
Afghanistan	1982	12881816	39.854	978.011439
Afghanistan	1987	13867957	40.822	852.395945
Afghanistan	1992	16317921	41.674	649.341395
Afghanistan	1997	22227415	41.763	635.341351
Afghanistan	2002	25268405	42.129	726.734055
Afghanistan	2007	31889923	43.828	974.580338

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

Generally, it is advisable to keep unique indices. But it also depends on the use-case.

How can we reset our indices back to integers?

In [60]:
```
df.reset_index()
```

Out[60]:

	index	Country	year	population	life_exp	gdp_cap
0	1	Afghanistan	1952	8425333	28.801	779.445314
1	2	Afghanistan	1957	9240934	30.332	820.853030
2	3	Afghanistan	1962	10267083	31.997	853.100710
3	4	Afghanistan	1967	11537966	34.020	836.197138
4	5	Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...	...	...
1699	1700	Zimbabwe	1987	9216418	62.351	706.157306
1700	1701	Zimbabwe	1992	10704340	60.377	693.420786
1701	1702	Zimbabwe	1997	11404948	46.809	792.449960
1702	1703	Zimbabwe	2002	11926563	39.989	672.038623
1703	1704	Zimbabwe	2007	12311143	43.487	469.709298

1704 rows × 6 columns

Notice that it's creating a new column `index`.

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

In [61]:
```
df.reset_index(drop=True) # by using drop=True we can prevent creation of a
```

Out[61]:

	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

Great!

Now let's do this in place.

In [62]:
```
df.reset_index(drop=True, inplace=True)
```

In [63]:
```
df
```

Out[63]:

	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

In [ ]:
```

```

Pandas 2

Content

- Working with both rows & columns
- Handling duplicate records
- Pandas built-in operations
  - Aggregate functions
  - Sorting values
- Concatenating DataFrames
- Merging DataFrames

Working with rows & columns together

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
df = pd.read_csv('mckinsey.csv')
df
```

Out[2]:

	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

How can we add a row to our dataframe?

There are multiple ways to do this.

- `concat()`
- `loc/iloc`

How can we do add a row using the `concat()` method?

In [3]:
```
new_row = {'country': 'India', 'year': 2000,'population':13500000, 'continen

df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
df
```

Out[3]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000

1705 rows × 6 columns

Why are we using `ignore_index=True` ?

- This parameter tells Pandas to ignore the existing index and create a new one based on the length of the resulting DataFrame.

Perfect! Our row is now added at the bottom of the dataframe.

Note:

- `concat()` doesn't mutate the the dataframe.
- It does not change the DataFrame, but returns a new DataFrame with the appended row.

Another method would be by **using loc**.

We will need to provide the position at which we want to add the new row.

What do you think this positional value would be?

- `len(df.index)` since we will add the new row at the end.

For this method we only need to insert the values of columns in the respective manner.

In [4]:
```
new_row = {'country': 'India', 'year': 2000,'population':13500000, 'continen
new_row_val = list(new_row.values())
new_row_val
```

Out[4]:
```
['India', 2000, 13500000, 'Asia', 37.08, 900.23]
```

In [5]:
```
df.loc[len(df.index)] = new_row_val
df
```

Out[5]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	India	2000	13500000	Asia	37.080	900.230000

1706 rows × 6 columns

The new row was added but the data has been duplicated.

What you can infer from last two duplicate rows?

- DataFrame allow us to feed duplicate rows in the data.

Now, can we also use `iloc` ?

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

In [6]:
```
df.iloc[len(df.index)-1] = ['Japan', 1000, 1350000, 'Asia', 37.08, 100.23]
df
```

Out[6]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1706 rows × 6 columns

What if we try to add the row with a new index?

In [7]:
```
df.iloc[len(df.index)] = ['India', 2000, 13500000, 'Asia', 37.08, 900.23]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 df.iloc[len(df.index)] = ['India', 2000, 13500000, 'Asia', 37.08, 9
00.23]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:815,
in _LocationIndexer.__setitem__(self, key, value)
    813     key = com.apply_if_callable(key, self.obj)
    814 indexer = self._get_setitem_indexer(key)
--> 815 self._has_valid_setitem_indexer(key)
    817 iloc = self if self.name == "iloc" else self.obj.iloc
    818 iloc._setitem_with_indexer(indexer, value, self.name)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1518,
in _iLocIndexer._has_valid_setitem_indexer(self, indexer)
   1516 elif is_integer(i):
   1517     if i >= len(ax):
-> 1518         raise IndexError("iloc cannot enlarge its target object")
   1519 elif isinstance(i, dict):
   1520     raise IndexError("iloc cannot enlarge its target object")

IndexError: iloc cannot enlarge its target object
```

Why are we getting an error?

- For using `iloc` to add a row, the dataframe must already have a row in that position.
- If a row is not available, you'll see this `IndexError`.

Note: When using the `loc[]` attribute, it's not mandatory that a row already exists with a specific label.

What if we want to delete a row?

- use `df.drop()`

If you remember we specified axis=1 for columns.

We can modify this - `axis=0` for rows.

Does `drop()` method uses positional indices or labels?

- We had to specify column title.
- So **`drop()` uses labels**, NOT positional indices.

\ Let's drop the row with label 3.

In [8]:
```
df
```

Out[8]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1706 rows × 6 columns

In [9]:
```
df = df.drop(3, axis=0)
df
```

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
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1705 rows × 6 columns

We can see that the **row with label 3 is deleted**.

We now have **rows with labels 0, 1, 2, 4, 5, ...**

`df.loc[4]` and `df.iloc[4]` will give different results.

In [10]:
```
df.loc[4] # The 4th row is printed
```

Out[10]:
```
country        Afghanistan
year                  1972
population        13079460
continent             Asia
life_exp            36.088
gdp_cap         739.981106
Name: 4, dtype: object
```

In [11]:
```
df.iloc[4] # The 5th row is printed
```

Out[11]:
```
country        Afghanistan
year                  1977
population        14880372
continent             Asia
life_exp            38.438
gdp_cap          786.11336
Name: 5, dtype: object
```

Why did this happen?

It is because the `loc` function selects rows using row labels (0,1,2,4,..) whereas the `iloc` function selects rows using their integer positions (staring from 0 and +1 for each row).

So for `iloc`, the 5th row starting from 0 index was printed.

How can we drop multiple rows?

In [12]:
```
df.drop([1, 2, 4], axis=0) # drops rows with labels 1, 2, 4
```

Out[12]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1702 rows × 6 columns

Let's reset our indices now.

In [13]:
```
df.reset_index(drop=True,inplace=True) # since we removed a row earlier, we
df
```

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
3	Afghanistan	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1700	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1701	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1703	India	2000	13500000	Asia	37.080	900.230000
1704	Japan	1000	1350000	Asia	37.080	100.230000

1705 rows × 6 columns

Handling duplicate records

If you remember, the last two rows were duplicates.

How can we deal with these duplicate rows?

Let's create some more duplicate rows to understand this.

In [14]:
```
df.loc[len(df.index)] = ['India', 2000, 13500000, 'Asia', 37.08, 900.23]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000, 'Asia', 80.00,500.00]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000, 'Asia', 80.00,500.00]
df.loc[len(df.index)] = ['India',2000 ,13500000, 'Asia', 80.00,900.23]
df
```

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	10267083	Asia	31.997	853.100710
3	Afghanistan	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1704	Japan	1000	1350000	Asia	37.080	100.230000
1705	India	2000	13500000	Asia	37.080	900.230000
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1709 rows × 6 columns

How to check for duplicate rows?

- We use `duplicated()` method on the DataFrame.

In [15]:
```
df.duplicated()
```

Out[15]:
```
0       False
1       False
2       False
3       False
4       False
        ...
1704    False
1705     True
1706    False
1707     True
1708    False
Length: 1709, dtype: bool
```

It gives True if an entire row is identical to the previous row.

However, it is not practical to see a list of True and False.

We can the `loc` data selector to extract those duplicate rows.

In [16]:
```
df.loc[df.duplicated()]
```

Out[16]:

	country	year	population	continent	life_exp	gdp_cap
1705	India	2000	13500000	Asia	37.08	900.23
1707	Sri Lanka	2022	130000000	Asia	80.00	500.00

How do we get rid of these duplicate rows?

- We can use the `drop_duplicates()` function.

In [17]:
```
df.drop_duplicates()
```

Out[17]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1703	India	2000	13500000	Asia	37.080	900.230000
1704	Japan	1000	1350000	Asia	37.080	100.230000
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1707 rows × 6 columns

But how do we decide among all duplicate rows which ones to keep?

Here we can use the `keep` argument.

It has only three distinct values -

- `first`
- `last`
- `False`

The default is 'first'.

If `first`, this considers first value as unique and rest of the identical values as duplicate.

In [18]:
```
df.drop_duplicates(keep='first')
```

Out[18]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1703	India	2000	13500000	Asia	37.080	900.230000
1704	Japan	1000	1350000	Asia	37.080	100.230000
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1707 rows × 6 columns

If `last`, this considers last value as unique and rest of the identical values as duplicate.

In [19]:
```
df.drop_duplicates(keep='last')
```

Out[19]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1704	Japan	1000	1350000	Asia	37.080	100.230000
1705	India	2000	13500000	Asia	37.080	900.230000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1707 rows × 6 columns

If `False`, this considers all the identical values as duplicates.

In [20]:
```
df.drop_duplicates(keep=False)
```

Out[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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1700	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1701	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1704	Japan	1000	1350000	Asia	37.080	100.230000
1708	India	2000	13500000	Asia	80.000	900.230000

1705 rows × 6 columns

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

We can use the `subset` argument to mention the list of columns which we want to use.

In [21]:
```
df.drop_duplicates(subset=['country'],keep='first')
```

Out[21]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
11	Albania	1952	1282697	Europe	55.230	1601.056136
23	Algeria	1952	9279525	Africa	43.077	2449.008185
35	Angola	1952	4232095	Africa	30.015	3520.610273
47	Argentina	1952	17876956	Americas	62.485	5911.315053
...	...	...	...	...	...	...
1643	Vietnam	1952	26246839	Asia	40.412	605.066492
1655	West Bank and Gaza	1952	1030585	Asia	43.160	1515.592329
1667	Yemen, Rep.	1952	4963829	Asia	32.548	781.717576
1679	Zambia	1952	2672000	Africa	42.038	1147.388831
1691	Zimbabwe	1952	3080907	Africa	48.451	406.884115

142 rows × 6 columns

Slicing the DataFrame

How can we slice the dataframe into, say first 4 rows and first 3 columns?

- We can use `iloc`

In [22]:
```
df.iloc[0:4, 0:3]
```

Out[22]:

	country	year	population
0	Afghanistan	1952	8425333
1	Afghanistan	1957	9240934
2	Afghanistan	1962	10267083
3	Afghanistan	1972	13079460

Pass in 2 different ranges for slicing - **one for row** and **one for column**, just like Numpy.

Recall, `iloc` doesn't include the end index while slicing.

Can we do the same thing with `loc`?

In [23]:
```
df.loc[1:5, 1:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 df.loc[1:5, 1:4]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1067,
in _LocationIndexer.__getitem__(self, key)
   1065     if self._is_scalar_access(key):
   1066         return self.obj._get_value(*key, takeable=self._takeable)
-> 1067     return self._getitem_tuple(key)
   1068 else:
   1069     # we by definition only have the 0th axis
   1070     axis = self.axis or 0

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1256,
in _LocIndexer._getitem_tuple(self, tup)
   1253 if self._multi_take_opportunity(tup):
   1254     return self._multi_take(tup)
-> 1256 return self._getitem_tuple_same_dim(tup)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:924,
in _LocationIndexer._getitem_tuple_same_dim(self, tup)
    921 if com.is_null_slice(key):
    922     continue
--> 924 retval = getattr(retval, self.name)._getitem_axis(key, axis=i)
    925 # We should never have retval.ndim < self.ndim, as that should
    926 #  be handled by the _getitem_lowerdim call above.
    927 assert retval.ndim == self.ndim

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1290,
in _LocIndexer._getitem_axis(self, key, axis)
   1288 if isinstance(key, slice):
   1289     self._validate_key(key, axis)
-> 1290     return self._get_slice_axis(key, axis=axis)
   1291 elif com.is_bool_indexer(key):
   1292     return self._getbool_axis(key, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1324,
in _LocIndexer._get_slice_axis(self, slice_obj, axis)
   1321     return obj.copy(deep=False)
   1323 labels = obj._get_axis(axis)
-> 1324 indexer = labels.slice_indexer(slice_obj.start, slice_obj.stop, sli
ce_obj.step)
   1326 if isinstance(indexer, slice):
   1327     return self.obj._slice(indexer, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
559, in Index.slice_indexer(self, start, end, step, kind)
   6516 """
   6517 Compute the slice indexer for input labels and step.
   6518 
   (...)
   6555 slice(1, 3, None)
   6556 """
   6557 self._deprecated_arg(kind, "kind", "slice_indexer")
-> 6559 start_slice, end_slice = self.slice_locs(start, end, step=step)
   6561 # return a slice
   6562 if not is_scalar(start_slice):

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
767, in Index.slice_locs(self, start, end, step, kind)
   6765 start_slice = None
   6766 if start is not None:
-> 6767     start_slice = self.get_slice_bound(start, "left")
```

```
   6768 if start_slice is None:
   6769     start_slice = 0

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
676, in Index.get_slice_bound(self, label, side, kind)
   6672 original_label = label
   6674 # For datetime indices label may be a string that has to be convert
ed
   6675 # to datetime boundary according to its resolution.
-> 6676 label = self._maybe_cast_slice_bound(label, side)
   6678 # we need to look up the label
   6679 try:

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
623, in Index._maybe_cast_slice_bound(self, label, side, kind)
   6618 # We are a plain index here (sub-class override this method if they
   6619 # wish to have special treatment for floats/ints, e.g. Float64Index
and
   6620 # datetimelike Indexes
   6621 # reject them, if index does not contain label
   6622 if (is_float(label) or is_integer(label)) and label not in self:
-> 6623     raise self._invalid_indexer("slice", label)
   6625 return label

TypeError: cannot do slice indexing on Index with these indexers [1] of typ
e int
```

Why does slicing using indices doesn't work with `loc` ?

Recall, we need to work with explicit labels while using `loc`.

In [24]:
```
df.loc[1:5, ['country','life_exp']]
```

Out[24]:

	country	life_exp
1	Afghanistan	30.332
2	Afghanistan	31.997
3	Afghanistan	36.088
4	Afghanistan	38.438
5	Afghanistan	39.854

In `loc`, we can mention ranges using column labels as well.

In [25]:
```
df.loc[1:5, 'year':'life_exp']
```

Out[25]:

	year	population	continent	life_exp
1	1957	9240934	Asia	30.332
2	1962	10267083	Asia	31.997
3	1972	13079460	Asia	36.088
4	1977	14880372	Asia	38.438
5	1982	12881816	Asia	39.854

How can we get specific rows and columns?

In [26]:
```
df.iloc[[0,10,100], [0,2,3]]
```

Out[26]:

	country	population	continent
0	Afghanistan	8425333	Asia
10	Afghanistan	31889923	Asia
100	Bangladesh	80428306	Asia

We pass in those **specific indices packed in `[]`**,

Can we do step slicing? Yes!

In [27]:
```
df.iloc[1:10:2]
```

Out[27]:

	country	year	population	continent	life_exp	gdp_cap
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
3	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1982	12881816	Asia	39.854	978.011439
7	Afghanistan	1992	16317921	Asia	41.674	649.341395
9	Afghanistan	2002	25268405	Asia	42.129	726.734055

Does step slicing work for `loc` too? Yes!

In [28]:
```
df.loc[1:10:2]
```

Out[28]:

	country	year	population	continent	life_exp	gdp_cap
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
3	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1982	12881816	Asia	39.854	978.011439
7	Afghanistan	1992	16317921	Asia	41.674	649.341395
9	Afghanistan	2002	25268405	Asia	42.129	726.734055

Pandas built-in operations

Aggregate functions

Let's select the feature `'life_exp'` -

In [29]:
```
le = df['life_exp']
le
```

Out[29]:
```
0       28.801
1       30.332
2       31.997
3       36.088
4       38.438
         ...
1704    37.080
1705    37.080
1706    80.000
1707    80.000
1708    80.000
Name: life_exp, Length: 1709, dtype: float64
```

How can we find the mean of the column `life_exp` ?

In [30]:
```
le.mean()
```

Out[30]:
```
59.486053060269164
```

What other operations can we do?

- `sum()`
- `count()`
- `min()`
- `max()`

... and so on

Note: We can see more methods by **pressing "tab" after `le.`**

In [31]:
```
le.sum()
```

Out[31]:
```
101661.66468
```

In [32]:
```
le.count()
```

Out[32]:
```
1709
```

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

In [33]:
```
le.sum() / le.count()
```

Out[33]:
```
59.486053060269164
```

It gives us the **mean/average** of life expectancy.

Sorting Values

If you notice, the `life_exp` column is not sorted.

How can we perform sorting in Pandas?

In [34]:
```
df.sort_values(['life_exp'])
```

Out[34]:

	country	year	population	continent	life_exp	gdp_cap
1291	Rwanda	1992	7290203	Africa	23.599	737.068595
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
551	Gambia	1952	284320	Africa	30.000	485.230659
35	Angola	1952	4232095	Africa	30.015	3520.610273
1343	Sierra Leone	1952	2143249	Africa	30.331	879.787736
...	...	...	...	...	...	...
1486	Switzerland	2007	7554661	Europe	81.701	37506.419070
694	Iceland	2007	301931	Europe	81.757	36180.789190
801	Japan	2002	127065841	Asia	82.000	28604.591900
670	Hong Kong, China	2007	6980412	Asia	82.208	39724.978670
802	Japan	2007	127467972	Asia	82.603	31656.068060

1709 rows × 6 columns

Rows get sorted **based on values in `life_exp` column**.

By default, values are sorted in **ascending order**.

How can we sort the rows in descending order?

In [35]:

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

Out[35]:

	country	year	population	continent	life_exp	gdp_cap
802	Japan	2007	127467972	Asia	82.603	31656.068060
670	Hong Kong, China	2007	6980412	Asia	82.208	39724.978670
801	Japan	2002	127065841	Asia	82.000	28604.591900
694	Iceland	2007	301931	Europe	81.757	36180.789190
1486	Switzerland	2007	7554661	Europe	81.701	37506.419070
...	...	...	...	...	...	...
1343	Sierra Leone	1952	2143249	Africa	30.331	879.787736
35	Angola	1952	4232095	Africa	30.015	3520.610273
551	Gambia	1952	284320	Africa	30.000	485.230659
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1291	Rwanda	1992	7290203	Africa	23.599	737.068595

1709 rows × 6 columns

Can we perform sorting on multiple columns? Yes!

In [36]:

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

Out[36]:

	country	year	population	continent	life_exp	gdp_cap
1704	Japan	1000	1350000	Asia	37.080	100.230000
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
551	Gambia	1952	284320	Africa	30.000	485.230659
35	Angola	1952	4232095	Africa	30.015	3520.610273
1343	Sierra Leone	1952	2143249	Africa	30.331	879.787736
...	...	...	...	...	...	...
694	Iceland	2007	301931	Europe	81.757	36180.789190
670	Hong Kong, China	2007	6980412	Asia	82.208	39724.978670
802	Japan	2007	127467972	Asia	82.603	31656.068060
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000

1709 rows × 6 columns

What exactly happened here?

- Rows were **first sorted** based on `'year'`
- Then, **rows with same values of** `'year'` were sorted based on `'lifeExp'`

In [37]:
```
from IPython.display import Image
Image(filename='sort.png')
```

Out[37]:

```
df3 = df.sort_values(["weight", "height"])
df3.head(10)
```

	name	age	height	weight	shirt_size
2	Rafael	83	161	50	M
6	Jacob	29	178	63	L
0	Ron	30	153	69	S
3	Karl-Hans	34	169	69	L
5	Ron	55	172	85	L
4	Freddy	20	169	86	S
1	Jacob	24	153	89	M

For same 'weight', 'height' is sorted in ascending order.

This way, we can do multi-level sorting of our data.

How can we have different sorting orders for different columns in multi-level sorting?

In [38]:
```
df.sort_values(['year', 'life_exp'], ascending=[False, True])
```

Out[38]:

	country	year	population	continent	life_exp	gdp_cap
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1462	Swaziland	2007	1133066	Africa	39.613	4513.480643
1042	Mozambique	2007	19951656	Africa	42.082	823.685621
1690	Zambia	2007	11746035	Africa	42.384	1271.211593
...	...	...	...	...	...	...
1463	Sweden	1952	7124673	Europe	71.860	8527.844662
1079	Netherlands	1952	10381988	Europe	72.130	8941.571858
683	Iceland	1952	147962	Europe	72.490	7267.688428
1139	Norway	1952	3327728	Europe	72.670	10095.421720
1704	Japan	1000	1350000	Asia	37.080	100.230000

1709 rows × 6 columns

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

Often times our data is separated into multiple tables, and we would require to work with them.

Let's see a mini use-case of `users` and `messages`.

`users` --> **Stores the user details - IDs** and **Names of users**

In [42]:
```
users = pd.DataFrame({"userid":[1, 2, 3], "name":["sharadh", "shahid", "khus
users
```

Out[42]:

	userid	name
0	1	sharadh
1	2	shahid
2	3	khusalli

`msgs` --> **Stores the messages** users have sent - **User IDs** and **Messages**

In [43]:
```
msgs = pd.DataFrame({"userid":[1, 1, 2, 4], "msg":['hmm', "acha", "theek ha
msgs
```

Out[43]:

	userid	msg
0	1	hmm
1	1	acha
2	2	theek hai
3	4	nice

Can we combine these 2 DataFrames to form a single DataFrame?

In [44]:
```
pd.concat([users, msgs])
```

Out[44]:

	userid	name	msg
0	1	sharadh	NaN
1	2	shahid	NaN
2	3	khusalli	NaN
0	1	NaN	hmm
1	1	NaN	acha
2	2	NaN	theek hai
3	4	NaN	nice

How exactly did `concat()` work?

- **By default**, `axis=0` (row-wise) for concatenation.
- **`userid`**, being same in both DataFrames, was **combined into a single column**.
  - First values of `users` dataframe were placed, with values of column `msg` as NaN
  - Then values of `msgs` dataframe were placed, with values of column `msg` as NaN
- The original indices of the rows were preserved.

How can we make the indices unique for each row?

In [45]:
```
pd.concat([users, msgs], ignore_index = True)
```

Out[45]:

	userid	name	msg
0	1	sharadh	NaN
1	2	shahid	NaN
2	3	khusalli	NaN
3	1	NaN	hmm
4	1	NaN	acha
5	2	NaN	theek hai
6	4	NaN	nice

How can we concatenate them horizontally?

In [46]:
```
pd.concat([users, msgs], axis=1)
```

Out[46]:

	userid	name	userid	msg
0	1.0	sharadh	1	hmm
1	2.0	shahid	1	acha
2	3.0	khusalli	2	theek hai
3	NaN	NaN	4	nice

As you can see here,

- Both the dataframes are combined horizontally (column-wise).
- It gives 2 columns with **different positional (implicit) index**, but **same label**.

Merging DataFrames

So far we have only concatenated but not merged data.

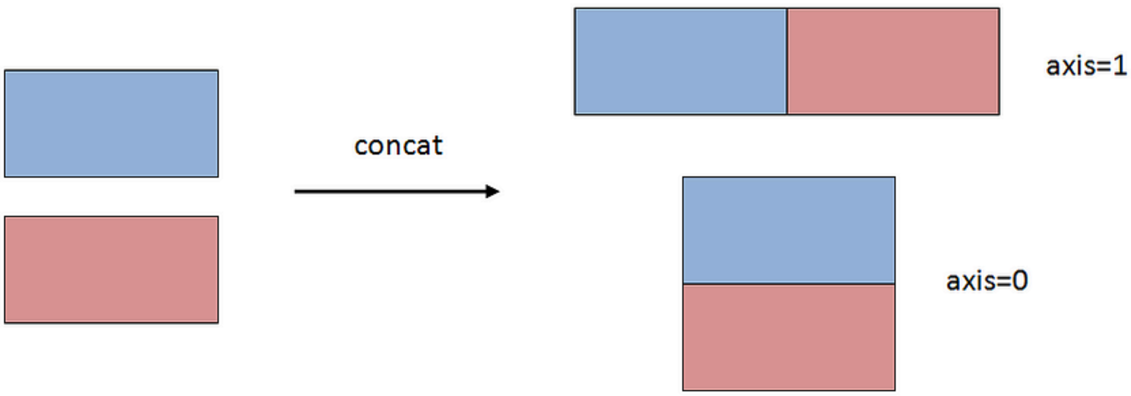
But what is the difference between `concat` and `merge`?

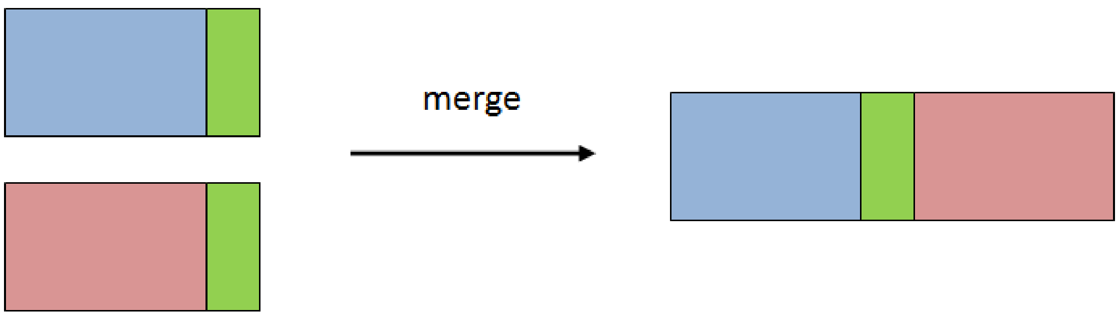
`concat`

- simply stacks multiple dataframes together along an axis.

`merge`

- combines dataframes in a **smart** way based on values in shared column(s).





How can we know the name of the person who sent a particular message?

We need information from **both the dataframes**.

So can we use `pd.concat()` for combining the dataframes? No.

In [47]:
```
pd.concat([users, msgs], axis=1)
```

Out[47]:

	userid	name	userid	msg
0	1.0	sharadh	1	hmm
1	2.0	shahid	1	acha
2	3.0	khusalli	2	theek hai
3	NaN	NaN	4	nice

What are the problems with here?

- `concat` simply **combined/stacked** the dataframe **horizontally**.
- If you notice, `userid 3` for **user** dataframe is stacked against `userid 2` for **msg** dataframe.
- This way of stacking doesn't help us gain any insights.

We need to **merge** the data.

How can we join the dataframes?

In [48]:
```
users.merge(msgs, on="userid")
```

Out[48]:

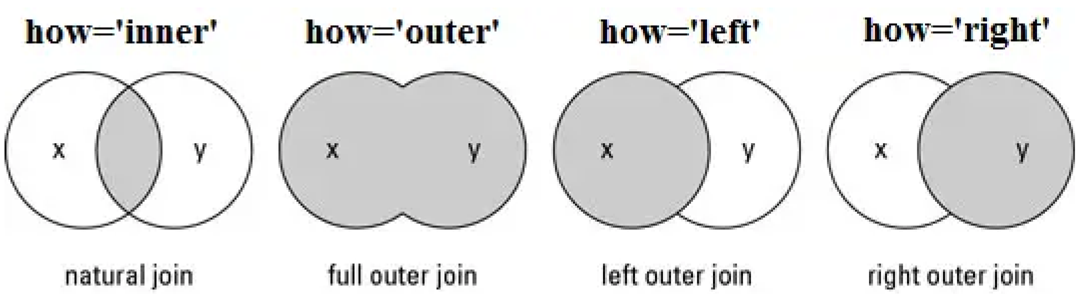
	userid	name	msg
0	1	sharadh	hmm
1	1	sharadh	acha
2	2	shahid	theek hai

Notice that `users` has a userid=3 but `msgs` does not.

- When we **merge** these dataframes, the **userid=3 is not included**.
- Similarly, **userid=4 is not present** in `users`, and thus **not included**.
- Only the userid **common in both dataframes** is shown.

What type of join is this? Inner Join

Remember joins from SQL?



The `on` parameter specifies the `key`, similar to `primary key` in SQL.

\ What join we want to use to get info of all the users and all the messages?

In [49]:
```
users.merge(msgs, on="userid", how="outer")
```

Out[49]:

	userid	name	msg
0	1	sharadh	hmm
1	1	sharadh	acha
2	2	shahid	theek hai
3	3	khusalli	NaN
4	4	NaN	nice

Note: All missing values are replaced with `NaN`.

What if we want the info of all the users in the dataframe?

In [50]:
```
users.merge(msgs, on="userid", how="left")
```

Out[50]:

	userid	name	msg
0	1	sharadh	hmm
1	1	sharadh	acha
2	2	shahid	theek hai
3	3	khusalli	NaN

Similarly, what if we want all the messages and info only for the users who sent a message?

In [51]:
```
users.merge(msgs, on="userid", how="right")
```

Out[51]:

	userid	name	msg
0	1	sharadh	hmm
1	1	sharadh	acha
2	2	shahid	theek hai
3	4	NaN	nice

`NaN` in **name** can be thought of as an anonymous message.

But sometimes, the column names might be different even if they contain the same data.

Let's rename our users column `userid` to `id`.

In [52]:
```
users.rename(columns = {"userid": "id"}, inplace=True)
users
```

Out[52]:

	id	name
0	1	sharadh
1	2	shahid
2	3	khusalli

Now, how can we merge the 2 dataframes when the `key` has a different value?

In [53]:
```
users.merge(msgs, left_on="id", right_on="userid")
```

Out[53]:

	id	name	userid	msg
0	1	sharadh	1	hmm
1	1	sharadh	1	acha
2	2	shahid	2	theek hai

Here,

- `left_on` : Specifies the **key of the 1st dataframe** (users).
- `right_on` : Specifies the **key of the 2nd dataframe** (msgs).

In [ ]:

Content

- Introduction to IMDB use case
    - Merging `movies` & `directors` datasets
    - IMDB data exploration (Post-read)
- `apply()`
- `groupby()`
    - Group based Aggregation
    - Group based Filtering
    - Group based Apply

IMDB Movies Data

- Imagine you are working as a Data Scientist for an analytics firm.
- Your task is to analyse some **movie trends** for a client.
- **IMDB** has an online database of information related to movies.

Here we have two CSV files -

- `movies.csv`
- `directors.csv`

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
movies = pd.read_csv('movies.csv')
movies.head()
```

Out[2]:

	Unnamed: 0	id	budget	popularity	revenue	title	vote_average	vote_c
0	0	43597	237000000	150	2787965087	Avatar	7.2	1
1	1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	'
2	2	43599	245000000	107	880674609	Spectre	6.3	'
3	3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	
4	5	43602	258000000	115	890871626	Spider-Man 3	5.9	:

So what kind of questions can we ask from this dataset?

- **Top 10 most popular movies**, using `popularity`.
- Find the **highest rated movies**, using `vote_average`.

- We can find number of **movies released per year**.
- Find **highest budget movies in a year** using both `budget` and `year`.

But can we ask more interesting/deeper questions?

- Do you think we can find the **most productive directors**?
- Which **directors produce high budget films**?
- **Highest and lowest rated movies for every month** in a particular year?

Notice that there's a column **Unnamed: 0** which represents nothing but the index of a row.

How to get rid of this `Unnamed: 0` col?

In [3]:
```
movies = pd.read_csv('movies.csv', index_col=0)
movies.head()
```

Out[3]:

	id	budget	popularity	revenue	title	vote_average	vote_count	direct
0	43597	237000000	150	2787965087	Avatar	7.2	11800	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	
2	43599	245000000	107	880674609	Spectre	6.3	4466	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	
5	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	

`index_col=0` explicitly states to treat the first column as the index.

The default value is `index_col=None`

In [4]:
```
movies.shape
```

Out[4]:
```
(1465, 11)
```

The `movies` dataframe contains 1465 rows and 11 columns.

In [5]:
```
directors = pd.read_csv('directors.csv', index_col=0)
directors.head()
```

Out[5]:

	director_name	id	gender
0	James Cameron	4762	Male
1	Gore Verbinski	4763	Male
2	Sam Mendes	4764	Male
3	Christopher Nolan	4765	Male
4	Andrew Stanton	4766	Male

In [6]:
```
directors.shape
```
Out[6]:
```
(2349, 3)
```

Merging `movies` & `directors` datasets

How can we know the details about the Director of a particular movie?

- We will have to merge these two datasets.

So on which column we should merge?

We will use the **ID** columns (representing unique directors) in both the datasets.

If you observe,

- `director_id` of movies are taken from `id` of directors.
- Thus, we can merge our dataframes based on these two columns as **keys**.

Before that, let's first check the number of unique directors in our `movies` dataset.

How do we get the number of unique directors in `movies` ?

In [7]:
```
movies['director_id'].nunique()
```
Out[7]:
```
199
```

Recall, we had learnt about `nunique()` earlier.

In [8]:
```
directors['id'].nunique()
```
Out[8]:
```
2349
```

Summary:

- `movies` dataset: 1465 rows, but only 199 unique directors
- `directors` dataset: 2349 unique directors (equal to the no. of rows)

What can we infer from this?

- The directors in `movies` data is a subset of directors in `directors` data.

How can we check if all `director_id` values are present in `id` ?

In [9]:
```
movies['director_id'].isin(directors['id'])
```

Out[9]:
```
0       True
1       True
2       True
3       True
5       True
        ...
4736    True
4743    True
4748    True
4749    True
4768    True
Name: director_id, Length: 1465, dtype: bool
```

The `isin()` method checks if a column contains the specified value(s).

How is `isin` different from Python's `in`?

- `in` works for **one element** at a time.
- `isin` does this for **all the values** in the column.

If you notice,

- This is like a **boolean mask**.
- It returns a dataframe similar to the original one.
- For rows with values of `director_id` present in `id`, it returns True, else False.

How can we check if there's any False here?

In [10]:
```
np.all(movies['director_id'].isin(directors['id']))
```

Out[10]:
```
True
```

Let's finally merge the two dataframes.

Do we need to keep **all the rows for movies**? Yes!

Do we need to keep **all the rows of directors**? No.

- Only the ones for which we have a corresponding row in `movies`.

So which `join` type do you think we should apply here?

- `LEFT` Join

In [11]:
```
data = movies.merge(directors, how='left', left_on='director_id',right_on='
data
```

Out[11]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	di
0	43597	237000000	150	2787965087	Avatar	7.2	11800	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	
2	43599	245000000	107	880674609	Spectre	6.3	4466	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	
1461	48370	27000	19	3151130	Clerks	7.4	755	
1462	48375	0	7	0	Rampage	6.0	131	
1463	48376	0	3	0	Slacker	6.4	77	
1464	48395	220000	14	2040920	El Mariachi	6.6	238	

1465 rows × 14 columns

Notice the two strange id columns - `id_x` and `id_y`.

What do you think these newly created columns are?

Since the columns with name `id` are present in both the dataframes,

- `id_x` represents **id values from movie df**
- `id_y` represents **id values from directors df**

Do you think any column is redundant here and can be dropped?

- `id_y` is redundant as it is the same as `director_id`
- But we don't require the `director_id` any further.

So we can simply drop these features -

In [12]:
```
data.drop(['director_id','id_y'], axis=1, inplace=True)
data.head()
```

Out[12]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007

Post-read

- IMDB data exploration

From here, we have the opportunity to delve into various aspects of the data, such as:

- Converting the revenue values into Millions of USD.
- Identifying the Top 5 most popular movies.

... and so on.

This task is for you to explore the data on your own.

`apply()`

- It is used apply a function along an axis of the DataFrame/Series.

Say we want to convert the data in `Gender` column into numerical format.

Basically,

- 0 for Male
- 1 for Female

How can we encode the values in the `Gender` column?

Let's first write a function to do it for a single value.

In [13]:
```
def encode(data):
  if data == "Male":
    return 0
  else:
    return 1
```

Now how can we apply this function to the whole column?

In [14]:
```
data['gender'] = data['gender'].apply(encode)
data
```

Out[14]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2(
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2(
2	43599	245000000	107	880674609	Spectre	6.3	4466	2(
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2(
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2(
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	1(
1461	48370	27000	19	3151130	Clerks	7.4	755	1(
1462	48375	0	7	0	Rampage	6.0	131	2(
1463	48376	0	3	0	Slacker	6.4	77	1(
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1(

1465 rows × 12 columns

Notice how this is similar to using `Vectorization` in Numpy.

How to apply a function on multiple columns?

Let's say we want to find the sum of `revenue` and `budget` per movie?

In [15]:
```
data[['revenue', 'budget']].apply(np.sum)
```

Out[15]:
```
revenue    209866997305
budget      70353617179
dtype: int64
```

We can pass multiple columns by packing them within `[]`.

But there's a mistake here. We wanted our results per movie (i.e. per row)

But we're getting the sum of the columns.

In [16]:
```
data[['revenue', 'budget']].apply(np.sum, axis=1)
```

Out[16]:
```
0       3024965087
1       1261000000
2       1125674609
3       1334939099
4       1148871626
           ...    
1460        321952
1461       3178130
1462             0
1463             0
1464       2260920
Length: 1465, dtype: int64
```

By setting the `axis=1`, every row of `revenue` was added to same row of `budget`.

What does this `axis` mean in apply?

- `axis=0` $\rightarrow$ It will apply to **each column**
- `axis=1` $\rightarrow$ It will apply to **each row**

Note that **by default, axis=0**.

Similarly, how can I find the `profit` per movie (revenue-budget)?

In [17]:
```
# We define a function to calculate profit

def prof(x):
  return x['revenue']-x['budget']
data['profit'] = data[['revenue', 'budget']].apply(prof, axis = 1)
data
```

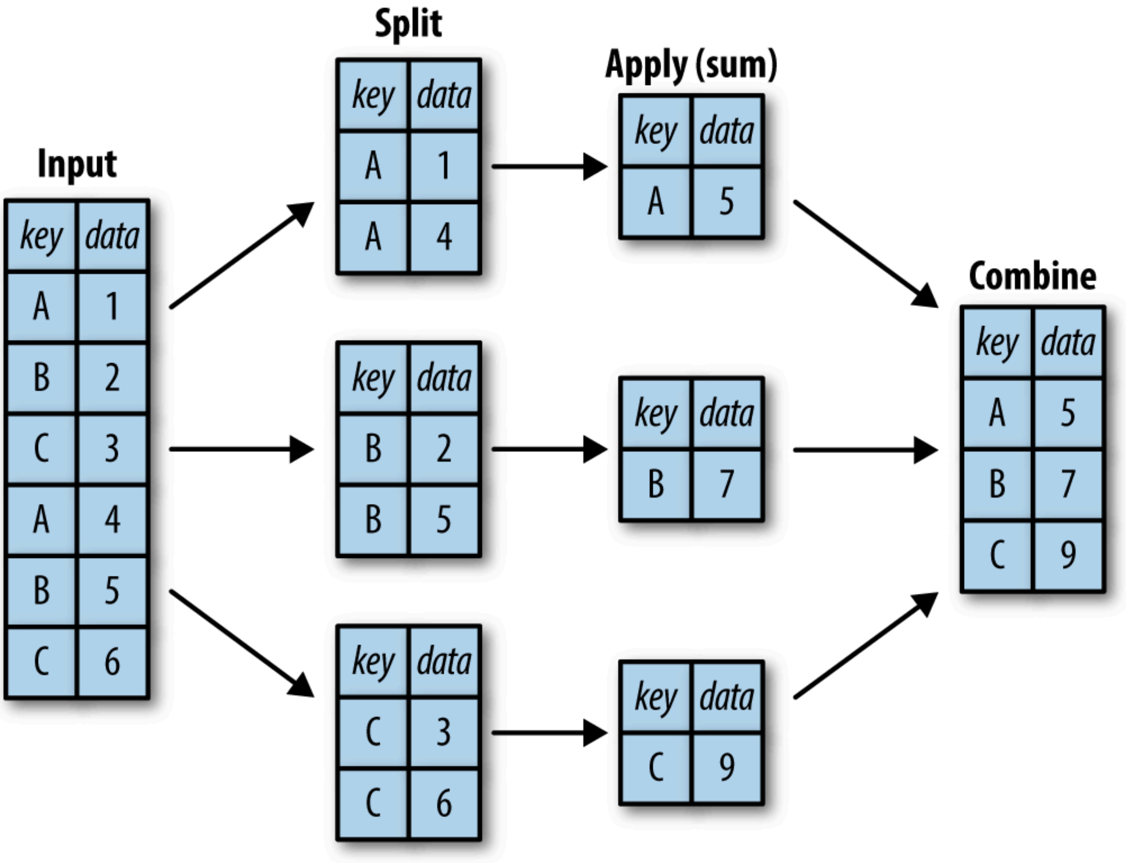
Out[17]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	2(
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2(
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 13 columns

What is Grouping?

In simple terms, we could understood it through - Split, Apply, Combine



1. **Split**: Breaking up and grouping a DataFrame depending on the value of the specified key.
2. **Apply**: Computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
3. **Combine**: Merging the results of these operations into an output array.

In [18]:
```
data.groupby('director_name')
```

Out[18]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x148f7cb10>

Notice,

- It's a **DataFrameGroupBy** type object
- **NOT a DataFrame** type object

What's the number of groups our data is divided into?

In [19]:
```
data.groupby('director_name').ngroups
```

Out[19]: 199

Based on this grouping, we can find which keys belong to which group.

In [20]:
```
data.groupby('director_name').groups
```

Out[20]:


```
{'Adam McKay': [176, 323, 366, 505, 839, 916], 'Adam Shankman': [265, 300,
350, 404, 458, 843, 999, 1231], 'Alejandro González Iñárritu': [106, 749, 1
015, 1034, 1077, 1405], 'Alex Proyas': [95, 159, 514, 671, 873], 'Alexander
Payne': [793, 1006, 1101, 1211, 1281], 'Andrew Adamson': [11, 43, 328, 501,
947], 'Andrew Niccol': [533, 603, 701, 722, 1439], 'Andrzej Bartkowiak': [3
49, 549, 754, 911, 924], 'Andy Fickman': [517, 681, 909, 926, 973, 1023],
'Andy Tennant': [314, 320, 464, 593, 676, 885], 'Ang Lee': [99, 134, 748, 8
40, 1089, 1110, 1132, 1184], 'Anne Fletcher': [610, 650, 736, 789, 1206],
'Antoine Fuqua': [310, 338, 424, 467, 576, 808, 818, 1105], 'Atom Egoyan':
[946, 1128, 1164, 1194, 1347, 1416], 'Barry Levinson': [313, 319, 471, 594,
878, 898, 1013, 1037, 1082, 1143, 1185, 1345, 1378], 'Barry Sonnenfeld': [1
3, 48, 90, 205, 591, 778, 783], 'Ben Stiller': [209, 212, 547, 562, 850],
'Bill Condon': [102, 307, 902, 1233, 1381], 'Bobby Farrelly': [352, 356, 48
1, 498, 624, 630, 654, 806, 928, 972, 1111], 'Brad Anderson': [1163, 1197,
1350, 1419, 1430], 'Brett Ratner': [24, 39, 188, 207, 238, 292, 405, 456, 9
20], 'Brian De Palma': [228, 255, 318, 439, 747, 905, 919, 1088, 1232, 126
1, 1317, 1354], 'Brian Helgeland': [512, 607, 623, 742, 933], 'Brian Levan
t': [418, 449, 568, 761, 860, 1003], 'Brian Robbins': [416, 441, 669, 962,
988, 1115], 'Bryan Singer': [6, 32, 33, 44, 122, 216, 297, 1326], 'Cameron
Crowe': [335, 434, 488, 503, 513, 698], 'Catherine Hardwicke': [602, 695, 7
24, 937, 1406, 1412], 'Chris Columbus': [117, 167, 204, 218, 229, 509, 656,
897, 996, 1086, 1129], 'Chris Weitz': [17, 500, 794, 869, 1202, 1267], 'Chr
istopher Nolan': [3, 45, 58, 59, 74, 565, 641, 1341], 'Chuck Russell': [17
7, 410, 657, 1069, 1097, 1339], 'Clint Eastwood': [369, 426, 447, 482, 490,
520, 530, 535, 645, 727, 731, 786, 787, 899, 974, 986, 1167, 1190, 1313],
'Curtis Hanson': [494, 579, 606, 711, 733, 1057, 1310], 'Danny Boyle': [52
7, 668, 1083, 1085, 1126, 1168, 1287, 1385], 'Darren Aronofsky': [113, 751,
1187, 1328, 1363, 1458], 'Darren Lynn Bousman': [1241, 1243, 1283, 1338, 14
40], 'David Ayer': [50, 273, 741, 1024, 1146, 1407], 'David Cronenberg': [5
41, 767, 994, 1055, 1254, 1268, 1334], 'David Fincher': [62, 213, 253, 383,
398, 478, 522, 555, 618, 785], 'David Gordon Green': [543, 862, 884, 927, 1
376, 1418, 1432, 1459], 'David Koepp': [443, 644, 735, 1041, 1209], 'David
Lynch': [583, 1161, 1264, 1340, 1456], 'David O. Russell': [422, 556, 609,
896, 982, 989, 1229, 1304], 'David R. Ellis': [582, 634, 756, 888, 934], 'D
avid Zucker': [569, 619, 965, 1052, 1175], 'Dennis Dugan': [217, 260, 267,
293, 303, 718, 780, 977, 1247], 'Donald Petrie': [427, 507, 570, 649, 858,
894, 1106, 1331], 'Doug Liman': [52, 148, 251, 399, 544, 1318, 1451], 'Edwa
rd Zwick': [92, 182, 346, 566, 791, 819, 825], 'F. Gary Gray': [308, 402, 4
91, 523, 697, 833, 1272, 1380], 'Francis Ford Coppola': [487, 559, 622, 64
6, 772, 1076, 1155, 1253, 1312], 'Francis Lawrence': [63, 72, 109, 120, 67
9], 'Frank Coraci': [157, 249, 275, 451, 577, 599, 963], 'Frank Oz': [193,
355, 473, 580, 712, 813, 987], 'Garry Marshall': [329, 496, 528, 571, 784,
893, 1029, 1169], 'Gary Fleder': [518, 667, 689, 867, 981, 1165], 'Gary Win
ick': [258, 797, 798, 804, 1454], 'Gavin O'Connor': [820, 841, 939, 953, 14
44], 'George A. Romero': [250, 1066, 1096, 1278, 1367, 1396], 'George Cloon
ey': [343, 450, 831, 966, 1302], 'George Miller': [78, 103, 233, 287, 1250,
1403, 1450], 'Gore Verbinski': [1, 8, 9, 107, 119, 633, 1040], 'Guillermo d
el Toro': [35, 252, 419, 486, 1118], 'Gus Van Sant': [595, 1018, 1027, 115
9, 1240, 1311, 1398], 'Guy Ritchie': [124, 215, 312, 1093, 1225, 1269, 142
0], 'Harold Ramis': [425, 431, 558, 586, 788, 1137, 1166, 1325], 'Ivan Reit
man': [274, 643, 816, 883, 910, 935, 1134, 1242], 'James Cameron': [0, 19,
170, 173, 344, 1100, 1320], 'James Ivory': [1125, 1152, 1180, 1291, 1293, 1
390, 1397], 'James Mangold': [140, 141, 557, 560, 829, 845, 958, 1145], 'Ja
mes Wan': [30, 617, 1002, 1047, 1337, 1417, 1424], 'Jan de Bont': [155, 22
4, 231, 270, 781], 'Jason Friedberg': [812, 1010, 1012, 1014, 1036], 'Jason
Reitman': [792, 1092, 1213, 1295, 1299], 'Jaume Collet-Serra': [516, 540, 6
40, 725, 1011, 1189], 'Jay Roach': [195, 359, 389, 397, 461, 703, 859, 107
2], 'Jean-Pierre Jeunet': [423, 485, 605, 664, 765], 'Joe Dante': [284, 52
5, 638, 1226, 1298, 1428], 'Joe Wright': [85, 432, 553, 803, 814, 855], 'Jo
el Coen': [428, 670, 691, 707, 721, 889, 906, 980, 1157, 1238, 1305], 'Joel
Schumacher': [128, 184, 348, 484, 572, 614, 652, 764, 876, 886, 1108, 1230,
1280], 'John Carpenter': [537, 663, 686, 861, 938, 1028, 1080, 1102, 1329,
1371], 'John Glen': [601, 642, 801, 847, 864], 'John Landis': [524, 868, 12
76, 1384, 1435], 'John Madden': [457, 882, 1020, 1249, 1257], 'John McTiern
```

```
an': [127, 214, 244, 351, 534, 563, 648, 782, 838, 1074], 'John Singleton':
[294, 489, 732, 796, 1120, 1173, 1316], 'John Whitesell': [499, 632, 763, 1
119, 1148], 'John Woo': [131, 142, 264, 371, 420, 675, 1182], 'Jon Favrea
u': [46, 54, 55, 382, 759, 1346], 'Jon M. Chu': [100, 225, 810, 1099, 118
6], 'Jon Turteltaub': [64, 180, 372, 480, 760, 846, 1171], 'Jonathan Demm
e': [277, 493, 1000, 1123, 1215], 'Jonathan Liebesman': [81, 143, 339, 111
7, 1301], 'Judd Apatow': [321, 710, 717, 865, 881], 'Justin Lin': [38, 123,
246, 1437, 1447], 'Kenneth Branagh': [80, 197, 421, 879, 1094, 1277, 1288],
'Kenny Ortega': [412, 852, 1228, 1315, 1365], 'Kevin Reynolds': [53, 502, 6
39, 1019, 1059], ...}
```

What if we want to extract data of a particular group from this list?

In [21]:
```
data.groupby('director_name').get_group('Alexander Payne')
```

Out[21]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	ye
793	45163	30000000	19	105834556	About Schmidt	6.7	362	20
1006	45699	20000000	40	177243185	The Descendants	6.7	934	2(
1101	46004	16000000	23	109502303	Sideways	6.9	478	20
1211	46446	12000000	29	17654912	Nebraska	7.4	636	2(
1281	46813	0	13	0	Election	6.7	270	19

How can we find the count of movies by each director?

In [23]:
```
data.groupby('director_name')['title'].count()
```

Out[23]:
```
director_name
Adam McKay                     6
Adam Shankman                  8
Alejandro González Iñárritu    6
Alex Proyas                    5
Alexander Payne                5
                              ..
Wes Craven                    10
Wolfgang Petersen              7
Woody Allen                   18
Zack Snyder                    7
Zhang Yimou                    6
Name: title, Length: 199, dtype: int64
```

How to find multiple aggregates of any feature?

Finding the very first year and the latest year a director released a movie i.e basically the **min** & **max** of the `year` column, grouped by `director_name`.

In [24]:
```
data.groupby(['director_name'])["year"].aggregate(['min', 'max'])
```

Out[24]:

	min	max
director_name		
Adam McKay	2004	2015
Adam Shankman	2001	2012
Alejandro González Iñárritu	2000	2015
Alex Proyas	1994	2016
Alexander Payne	1999	2013
...	...	...
Wes Craven	1984	2011
Wolfgang Petersen	1981	2006
Woody Allen	1977	2013
Zack Snyder	2004	2016
Zhang Yimou	2002	2014

199 rows × 2 columns

Note: We can also use `.agg` instead of `.aggregate` (both are same)

Group based Filtering

Group based filtering allows us to filter rows from each group by using conditional statements on each group rather than the whole dataframe.

How to find the details of the movies by high budget directors?

- Lets assume, high budget director -> any director with **atleast one movie with budget >100M**.

1. We can **group** the data by director and use `max` of the budget as aggregator.

In [25]:
```
data_dir_budget = data.groupby("director_name")["budget"].max().reset_index
data_dir_budget.head()
```

Out[25]:

	director_name	budget
0	Adam McKay	100000000
1	Adam Shankman	80000000
2	Alejandro González Iñárritu	135000000
3	Alex Proyas	140000000
4	Alexander Payne	30000000

1. We can **filter** out the director names with **max budget >100M**.

In [26]:
```
names = data_dir_budget.loc[data_dir_budget["budget"] >= 100, "director_name
```

1. Finally, we can filter out the details of the movies by these directors.

In [27]:
```
data.loc[data['director_name'].isin(names)]
```

Out[27]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 13 columns

Can we filter groups in a single go using Lambda functions? Yes!

In [28]:
```
data.groupby('director_name').filter(lambda x: x["budget"].max() >= 100)
```

Out[28]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2(
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2(
2	43599	245000000	107	880674609	Spectre	6.3	4466	2(
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2(
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2(
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	1(
1461	48370	27000	19	3151130	Clerks	7.4	755	1(
1462	48375	0	7	0	Rampage	6.0	131	2(
1463	48376	0	3	0	Slacker	6.4	77	1(
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1(

1465 rows × 13 columns

Notice what's happening here?

- We first group data by director and then use `groupby().filter` function.
- **Groups are filtered if they do not satisfy the boolean criterion** specified by the function.
- This is called **Group Based Filtering**.

Note:

- We are filtering the **groups** here and **not the rows**.
- The result is **not a groupby object** but regular **Pandas DataFrame** with the **filtered groups eliminated**.

Group based Apply

- applying a function on grouped objects

What if we want to do the transformation of a column using some column's agrregate

Let's say, we want to filter the risky movies whose budget was even higher than the average revenue of the director from his other movies.

We can subtract the average `revenue` of a director from `budget` column, for each director.

In [29]:
```
def func(x):
  # returns whether a movie is risky or not
  x["risky"] = x["budget"] - x["revenue"].mean() >= 0
  return x

data_risky = data.groupby("director_name", group_keys=False).apply(func)
data_risky
```

Out[29]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 14 columns

Note:

- Setting `group_keys=True`, keeps the group key in the returned dataset.
- This will be default in future versions of Pandas.
- Keep it as False if want the normal behaviour.

What did we do here?

- Defined a custom function.
- Grouped data according to `director_name`.
- Subtracted the mean of `budget` from `revenue`.
- Used apply with the custom function on the grouped data.

Now let's see if there are any risky movies -

In [30]:
```
data_risky.loc[data_risky["risky"]]
```

Out[30]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	ye
7	43608	200000000	107	586090727	Quantum of Solace	6.1	2965	20
12	43614	380000000	135	1045713802	Pirates of the Caribbean: On Stranger Tides	6.4	4948	2(
15	43618	200000000	37	310669540	Robin Hood	6.2	1398	2C
20	43624	209000000	64	303025485	Battleship	5.5	2114	2C
24	43630	210000000	3	459359555	X-Men: The Last Stand	6.3	3525	20
...	...	...	...	...	...	...	...	
1347	47224	5000000	7	3263585	The Sweet Hereafter	6.8	103	19
1349	47229	5000000	3	4842699	90 Minutes in Heaven	5.4	40	2C
1351	47233	5000000	6	0	Light Sleeper	5.7	15	19
1356	47263	15000000	10	0	Dying of the Light	4.5	118	2C
1383	47453	3500000	4	0	In the Name of the King III	3.3	19	2C

131 rows × 14 columns

In [32]:
```
data_risky[data_risky["risky"]]
```

Out[32]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	ye
7	43608	200000000	107	586090727	Quantum of Solace	6.1	2965	20
12	43614	380000000	135	1045713802	Pirates of the Caribbean: On Stranger Tides	6.4	4948	20
15	43618	200000000	37	310669540	Robin Hood	6.2	1398	20
20	43624	209000000	64	303025485	Battleship	5.5	2114	20
24	43630	210000000	3	459359555	X-Men: The Last Stand	6.3	3525	20
...	...	...	...	...	...	...	...	
1347	47224	5000000	7	3263585	The Sweet Hereafter	6.8	103	19
1349	47229	5000000	3	4842699	90 Minutes in Heaven	5.4	40	20
1351	47233	5000000	6	0	Light Sleeper	5.7	15	19
1356	47263	15000000	10	0	Dying of the Light	4.5	118	20
1383	47453	3500000	4	0	In the Name of the King III	3.3	19	20

131 rows × 14 columns

In [ ]:

Pandas 4

Content

- Multi-indexing
- Melting
  - `pd.melt()`
- Pivoting
  - `pd.pivot()`
  - `pd.pivot_table()`
- Binning
  - `pd.cut()`

Multi-Indexing

In [1]:
```
import pandas as pd
import numpy as np

movies = pd.read_csv('movies.csv', index_col=0)
directors = pd.read_csv('directors.csv', index_col=0)

data = movies.merge(directors, how='left', left_on='director_id',right_on='i
data.drop(['director_id','id_y'],axis=1,inplace=True)
```

Which director according to you should be considered as most productive?

- Should we decide based on the **number of movies** directed?
- Or take the **quality of the movies** into consideration as well?
- Or maybe look at the the **amount of business** the movie is doing?

To simplify, let's calculate who has directed maximum number of movies.

In [2]:
```
data.groupby(['director_name'])['title'].count().sort_values(ascending=False
```

Out[2]:
```
director_name
Steven Spielberg    26
Clint Eastwood      19
Martin Scorsese     19
Woody Allen         18
Robert Rodriguez    16
                    ..
Paul Weitz           5
John Madden          5
Paul Verhoeven       5
John Whitesell       5
Kevin Reynolds       5
Name: title, Length: 199, dtype: int64
```

`Steven Spielberg` has directed maximum number of movies.

But does it make `Steven` the most productive director?

- Chances are, he might be active for more years than the other directors.

Calculating the active years for every director?

- We can subtract both `min` and `max` of year.

In [3]:
```
data_agg = data.groupby(['director_name'])[["year", "title"]].aggregate({"ye
data_agg
```

Out[3]:

	year		title
	min	max	count
director_name			
Adam McKay	2004	2015	6
Adam Shankman	2001	2012	8
Alejandro González Iñárritu	2000	2015	6
Alex Proyas	1994	2016	5
Alexander Payne	1999	2013	5
...	...	...	...
Wes Craven	1984	2011	10
Wolfgang Petersen	1981	2006	7
Woody Allen	1977	2013	18
Zack Snyder	2004	2016	7
Zhang Yimou	2002	2014	6

199 rows × 3 columns

Notice,

- `director_name` column has turned into **row labels**.
- There are multiple levels for the column names.

This is called a **Multi-index DataFrame**.

- It can have **multiple indexes along a dimension**.
  - The no. of dimensions remain same though.
- Multi-level indexes are **possible both for rows and columns**.

In [4]:
```
data_agg.columns
```

Out[4]:
```
MultiIndex([( 'year',   'min'),
            ( 'year',   'max'),
            ('title', 'count')],
           )
```

The level-1 column names are `year` and `title`.

What would happen if we print the column `year` of this multi-index dataframe?

In [5]:
```
data_agg["year"]
```

Out[5]:

	min	max
director_name		
Adam McKay	2004	2015
Adam Shankman	2001	2012
Alejandro González Iñárritu	2000	2015
Alex Proyas	1994	2016
Alexander Payne	1999	2013
...	...	...
Wes Craven	1984	2011
Wolfgang Petersen	1981	2006
Woody Allen	1977	2013
Zack Snyder	2004	2016
Zhang Yimou	2002	2014

199 rows × 2 columns

How can we convert multi-level back to only one level of columns?

- e.g. `year_min`, `year_max`, `title_count`

In [6]:
```
data_agg = data.groupby(['director_name'])[["year","title"]].aggregate(
    {"year":['min', 'max'], "title": "count"})
```

In [7]:
```
data_agg.columns = ['_'.join(col) for col in data_agg.columns]
data_agg
```

Out[7]:

	year_min	year_max	title_count
director_name			
Adam McKay	2004	2015	6
Adam Shankman	2001	2012	8
Alejandro González Iñárritu	2000	2015	6
Alex Proyas	1994	2016	5
Alexander Payne	1999	2013	5
...	...	...	...
Wes Craven	1984	2011	10
Wolfgang Petersen	1981	2006	7
Woody Allen	1977	2013	18
Zack Snyder	2004	2016	7
Zhang Yimou	2002	2014	6

199 rows × 3 columns

Since these were tuples, we can just join them.

In [8]:
```
data.groupby('director_name')[['year', 'title']].aggregate(
    year_max=('year','max'),
    year_min=('year','min'),
    title_count=('title','count')
)
```

Out[8]:

director_name	year_max	year_min	title_count
Adam McKay	2015	2004	6
Adam Shankman	2012	2001	8
Alejandro González Iñárritu	2015	2000	6
Alex Proyas	2016	1994	5
Alexander Payne	2013	1999	5
...	...	...	...
Wes Craven	2011	1984	10
Wolfgang Petersen	2006	1981	7
Woody Allen	2013	1977	18
Zack Snyder	2016	2004	7
Zhang Yimou	2014	2002	6

199 rows × 3 columns

The columns look good, but we may want to turn back the row labels into a proper column as well.

Converting row labels into a column using `reset_index` -

In [9]:
```
data_agg.reset_index()
```

Out[9]:

	director_name	year_min	year_max	title_count
0	Adam McKay	2004	2015	6
1	Adam Shankman	2001	2012	8
2	Alejandro González Iñárritu	2000	2015	6
3	Alex Proyas	1994	2016	5
4	Alexander Payne	1999	2013	5
...	...	...	...	...
194	Wes Craven	1984	2011	10
195	Wolfgang Petersen	1981	2006	7
196	Woody Allen	1977	2013	18
197	Zack Snyder	2004	2016	7
198	Zhang Yimou	2002	2014	6

199 rows × 4 columns

Using the new features, can we find the most productive director?

1. First calculate how many years the director has been active.

In [10]:
```
data_agg["yrs_active"] = data_agg["year_max"] - data_agg["year_min"]
data_agg
```

Out[10]:

director_name	year_min	year_max	title_count	yrs_active
Adam McKay	2004	2015	6	11
Adam Shankman	2001	2012	8	11
Alejandro González Iñárritu	2000	2015	6	15
Alex Proyas	1994	2016	5	22
Alexander Payne	1999	2013	5	14
...	...	...	...	...
Wes Craven	1984	2011	10	27
Wolfgang Petersen	1981	2006	7	25
Woody Allen	1977	2013	18	36
Zack Snyder	2004	2016	7	12
Zhang Yimou	2002	2014	6	12

199 rows × 4 columns

1. Then calculate rate of directing movies by `title_count` / `yrs_active` .

In [11]:
```
data_agg["movie_per_yr"] = data_agg["title_count"] / data_agg["yrs_active"]
data_agg
```

Out[11]:

director_name	year_min	year_max	title_count	yrs_active	movie_per_yr
Adam McKay	2004	2015	6	11	0.545455
Adam Shankman	2001	2012	8	11	0.727273
Alejandro González Iñárritu	2000	2015	6	15	0.400000
Alex Proyas	1994	2016	5	22	0.227273
Alexander Payne	1999	2013	5	14	0.357143
...	...	...	...	...	...
Wes Craven	1984	2011	10	27	0.370370
Wolfgang Petersen	1981	2006	7	25	0.280000
Woody Allen	1977	2013	18	36	0.500000
Zack Snyder	2004	2016	7	12	0.583333
Zhang Yimou	2002	2014	6	12	0.500000

199 rows × 5 columns

1. Finally, sort the values.

In [12]:
```
data_agg.sort_values("movie_per_yr", ascending=False)
```

Out[12]:

director_name	year_min	year_max	title_count	yrs_active	movie_per_yr
Tyler Perry	2006	2013	9	7	1.285714
Jason Friedberg	2006	2010	5	4	1.250000
Shawn Levy	2002	2014	11	12	0.916667
Robert Rodriguez	1992	2014	16	22	0.727273
Adam Shankman	2001	2012	8	11	0.727273
...	...	...	...	...	...
Lawrence Kasdan	1985	2012	5	27	0.185185
Luc Besson	1985	2014	5	29	0.172414
Robert Redford	1980	2010	5	30	0.166667
Sidney Lumet	1976	2006	5	30	0.166667
Michael Apted	1980	2010	5	30	0.166667

199 rows × 5 columns

Conclusion:

- `Tyler Perry` turns out to be truly the most productive director.

PFizer data

For this topic we will be using data of few drugs being developed by **PFizer**.

Dataset: https://drive.google.com/file/d/173A59xh2mnpmljCCB9bhC4C5eP2lS6qZ/view?usp=sharing

What is the data about?

- Temperature (K)
- Pressure (P)

The data is recorded after an **interval of 1 hour** everyday to monitor the drug stability in a drug development test.

These data points are therefore used to **identify the optimal set of values of parameters** for the stability of the drugs.

In [13]:
```
data = pd.read_csv('Pfizer_1.csv')
data
```

Out[13]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	NaN	21.0	21.0	22
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	NaN	11.0	13.0	14
2	15-10-2020	docetaxel injection	Temperature	NaN	17.0	18.0	NaN	17.0	18
3	15-10-2020	docetaxel injection	Pressure	NaN	22.0	22.0	NaN	22.0	23
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	NaN	NaN	27.0	NaN	26
5	15-10-2020	ketamine hydrochloride	Pressure	8.0	NaN	NaN	7.0	NaN	9
6	16-10-2020	diltiazem hydrochloride	Temperature	34.0	35.0	36.0	36.0	37.0	38
7	16-10-2020	diltiazem hydrochloride	Pressure	18.0	19.0	20.0	21.0	22.0	23
8	16-10-2020	docetaxel injection	Temperature	46.0	47.0	NaN	48.0	48.0	49
9	16-10-2020	docetaxel injection	Pressure	23.0	24.0	NaN	25.0	26.0	27
10	16-10-2020	ketamine hydrochloride	Temperature	8.0	9.0	10.0	NaN	11.0	12
11	16-10-2020	ketamine hydrochloride	Pressure	12.0	12.0	13.0	NaN	15.0	15
12	17-10-2020	diltiazem hydrochloride	Temperature	20.0	19.0	19.0	18.0	17.0	16
13	17-10-2020	diltiazem hydrochloride	Pressure	3.0	4.0	4.0	4.0	6.0	8
14	17-10-2020	docetaxel injection	Temperature	12.0	13.0	14.0	15.0	16.0	17
15	17-10-2020	docetaxel injection	Pressure	20.0	22.0	22.0	22.0	22.0	23
16	17-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	16.0	17.0	18

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
17	17-10-	ketamine hydrochloride	Pressure	8.0	9.0	10.0	11.0	11.0	12

In [14]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18 entries, 0 to 17
Data columns (total 15 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Date       18 non-null     object 
 1   Drug_Name  18 non-null     object 
 2   Parameter  18 non-null     object 
 3   1:30:00    16 non-null     float64
 4   2:30:00    16 non-null     float64
 5   3:30:00    12 non-null     float64
 6   4:30:00    14 non-null     float64
 7   5:30:00    16 non-null     float64
 8   6:30:00    18 non-null     int64  
 9   7:30:00    16 non-null     float64
 10  8:30:00    14 non-null     float64
 11  9:30:00    16 non-null     float64
 12  10:30:00   18 non-null     int64  
 13  11:30:00   16 non-null     float64
 14  12:30:00   18 non-null     int64  
dtypes: float64(9), int64(3), object(3)
memory usage: 2.2+ KB
```

Melting

As we saw earlier, the dataset has **18 rows** and **15 columns**.

If you notice further, you'll see:

- The columns are `1:30:00`, `2:30:00`, `3:30:00`, ... so on.
- `Temperature` and `Pressure` of each date is in a separate row.

Can we restructure our data into a better format?

- Maybe we can have a column for `time`, with `timestamps` as the column value.

Where will the Temperature/Pressure values go?

- We can similarly create one column containing the values of these parameters.
- "Melt" the timestamp column into two columns** - timestamp and corresponding values

How can we restructure our data into having every row corresponding to a single reading?

In [15]:
```
pd.melt(data, id_vars=['Date', 'Parameter', 'Drug_Name'])
```

Out[15]:

	Date	Parameter	Drug_Name	variable	value
0	15-10-2020	Temperature	diltiazem hydrochloride	1:30:00	23.0
1	15-10-2020	Pressure	diltiazem hydrochloride	1:30:00	12.0
2	15-10-2020	Temperature	docetaxel injection	1:30:00	NaN
3	15-10-2020	Pressure	docetaxel injection	1:30:00	NaN
4	15-10-2020	Temperature	ketamine hydrochloride	1:30:00	24.0
...	...	...	...	...	...
211	17-10-2020	Pressure	diltiazem hydrochloride	12:30:00	14.0
212	17-10-2020	Temperature	docetaxel injection	12:30:00	23.0
213	17-10-2020	Pressure	docetaxel injection	12:30:00	28.0
214	17-10-2020	Temperature	ketamine hydrochloride	12:30:00	24.0
215	17-10-2020	Pressure	ketamine hydrochloride	12:30:00	15.0

216 rows × 5 columns

This converts our data from `wide` to `long` format.

Notice that the `id_vars` are set of variables which remain unmelted.

How does `pd.melt()` work?

- Pass in the **DataFrame**.
- Pass in the **column names that we don't want to melt**.

But we can provide better names to these new columns.

How can we rename the columns "variable" and "value" as per our original dataframe?

In [16]:
```
data_melt = pd.melt(data,id_vars = ['Date', 'Drug_Name', 'Parameter'],
            var_name = "time",
            value_name = 'reading')
data_melt
```

Out[16]:

	Date	Drug_Name	Parameter	time	reading
0	15-10-2020	diltiazem hydrochloride	Temperature	1:30:00	23.0
1	15-10-2020	diltiazem hydrochloride	Pressure	1:30:00	12.0
2	15-10-2020	docetaxel injection	Temperature	1:30:00	NaN
3	15-10-2020	docetaxel injection	Pressure	1:30:00	NaN
4	15-10-2020	ketamine hydrochloride	Temperature	1:30:00	24.0
...	...	...	...	...	...
211	17-10-2020	diltiazem hydrochloride	Pressure	12:30:00	14.0
212	17-10-2020	docetaxel injection	Temperature	12:30:00	23.0
213	17-10-2020	docetaxel injection	Pressure	12:30:00	28.0
214	17-10-2020	ketamine hydrochloride	Temperature	12:30:00	24.0
215	17-10-2020	ketamine hydrochloride	Pressure	12:30:00	15.0

216 rows × 5 columns

Conclusion:

- The labels of the timestamp columns are conviniently **melted into a single column** - `time`
- It retained all the values in `reading` column.
- The labels of columns such as `1:30:00`, `2:30:00` have now become categories of the `variable` column.
- The values from columns we are melting are stored in the `value` column.

Pivoting

Now suppose we want to convert our data back to the **wide format**.

The reason could be to maintain the structure for storing or some other purpose.

Notice,

- The variables `Date`, `Drug_Name` and `Parameter` will remain same.
- The column names will be extracted from the column `time`.
- The values will be extracted from the column `readings`.

How can we restructure our data back to the original wide format?

In [17]:
```
data_melt.pivot(index=['Date','Drug_Name','Parameter'],  # Columns used to r
                columns = 'time',                         # Column used to ma
                values='reading')                         # Column used for p
```

Out[17]:

Date	Drug_Name	Parameter	10:30:00	11:30:00	12:30:00	1:30:00	2:30:00	3:30:0(
15-10-2020	diltiazem hydrochloride	Pressure	18.0	19.0	20.0	12.0	13.0	NaN
		Temperature	20.0	20.0	21.0	23.0	22.0	NaN
	docetaxel injection	Pressure	26.0	29.0	28.0	NaN	22.0	22.(
		Temperature	23.0	25.0	25.0	NaN	17.0	18.(
	ketamine hydrochloride	Pressure	9.0	9.0	11.0	8.0	NaN	NaN
		Temperature	22.0	21.0	20.0	24.0	NaN	NaN
16-10-2020	diltiazem hydrochloride	Pressure	24.0	NaN	27.0	18.0	19.0	20.(
		Temperature	40.0	NaN	42.0	34.0	35.0	36.(
	docetaxel injection	Pressure	28.0	29.0	30.0	23.0	24.0	NaN
		Temperature	56.0	57.0	58.0	46.0	47.0	NaN
	ketamine hydrochloride	Pressure	16.0	17.0	18.0	12.0	12.0	13.(
		Temperature	13.0	14.0	15.0	8.0	9.0	10.(
17-10-2020	diltiazem hydrochloride	Pressure	11.0	13.0	14.0	3.0	4.0	4.(
		Temperature	14.0	11.0	10.0	20.0	19.0	19.(
	docetaxel injection	Pressure	28.0	29.0	28.0	20.0	22.0	22.(
		Temperature	21.0	22.0	23.0	12.0	13.0	14.(
	ketamine hydrochloride	Pressure	13.0	14.0	15.0	8.0	9.0	10.(
		Temperature	22.0	23.0	24.0	13.0	14.0	15.(

Notice that `pivot()` is the exact opposite of `melt()`.

We are getting **multiple indices** here, but we can get single index again using `reset_index()`.

In [18]:
```
data_melt.pivot(index=['Date','Drug_Name','Parameter'],
                columns = 'time',
                values='reading').reset_index()
```

Out[18]:

time	Date	Drug_Name	Parameter	10:30:00	11:30:00	12:30:00	1:30:00	2:30:00	3:3
0	15-10-2020	diltiazem hydrochloride	Pressure	18.0	19.0	20.0	12.0	13.0	
1	15-10-2020	diltiazem hydrochloride	Temperature	20.0	20.0	21.0	23.0	22.0	
2	15-10-2020	docetaxel injection	Pressure	26.0	29.0	28.0	NaN	22.0	
3	15-10-2020	docetaxel injection	Temperature	23.0	25.0	25.0	NaN	17.0	
4	15-10-2020	ketamine hydrochloride	Pressure	9.0	9.0	11.0	8.0	NaN	
5	15-10-2020	ketamine hydrochloride	Temperature	22.0	21.0	20.0	24.0	NaN	
6	16-10-2020	diltiazem hydrochloride	Pressure	24.0	NaN	27.0	18.0	19.0	
7	16-10-2020	diltiazem hydrochloride	Temperature	40.0	NaN	42.0	34.0	35.0	
8	16-10-2020	docetaxel injection	Pressure	28.0	29.0	30.0	23.0	24.0	
9	16-10-2020	docetaxel injection	Temperature	56.0	57.0	58.0	46.0	47.0	
10	16-10-2020	ketamine hydrochloride	Pressure	16.0	17.0	18.0	12.0	12.0	
11	16-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	8.0	9.0	
12	17-10-2020	diltiazem hydrochloride	Pressure	11.0	13.0	14.0	3.0	4.0	
13	17-10-2020	diltiazem hydrochloride	Temperature	14.0	11.0	10.0	20.0	19.0	
14	17-10-2020	docetaxel injection	Pressure	28.0	29.0	28.0	20.0	22.0	
15	17-10-2020	docetaxel injection	Temperature	21.0	22.0	23.0	12.0	13.0	
16	17-10-2020	ketamine hydrochloride	Pressure	13.0	14.0	15.0	8.0	9.0	

time	Date	Drug_Name	Parameter	10:30:00	11:30:00	12:30:00	1:30:00	2:30:00	3:3
17	17-10-	ketamine hydrochloride	Temperature	22.0	23.0	24.0	13.0	14.0	

In [19]:
```python
data_melt.head()
```

Out[19]:

	Date	Drug_Name	Parameter	time	reading
0	15-10-2020	diltiazem hydrochloride	Temperature	1:30:00	23.0
1	15-10-2020	diltiazem hydrochloride	Pressure	1:30:00	12.0
2	15-10-2020	docetaxel injection	Temperature	1:30:00	NaN
3	15-10-2020	docetaxel injection	Pressure	1:30:00	NaN
4	15-10-2020	ketamine hydrochloride	Temperature	1:30:00	24.0

Now if you notice,

- We are using 2 rows to log readings for a single experiment.

Can we further restructure our data into dividing the `Parameter` column into T/P?

- A format like `Date | time | Drug_Name | Pressure | Temperature` would be suitable.
- We want to **split one single column into multiple columns**.

How can we divide the `Parameter` column again?

In [20]:
```python
data_tidy = data_melt.pivot(index=['Date','time', 'Drug_Name'],
                            columns = 'Parameter',
                            values='reading')

data_tidy
```

Out[20]:

		Parameter	Pressure	Temperature
Date	**time**	**Drug_Name**		
15-10-2020	**10:30:00**	**diltiazem hydrochloride**	18.0	20.0
		docetaxel injection	26.0	23.0
		ketamine hydrochloride	9.0	22.0
	11:30:00	**diltiazem hydrochloride**	19.0	20.0
		docetaxel injection	29.0	25.0
...	**...**	**...**	...	...
17-10-2020	**8:30:00**	**docetaxel injection**	26.0	19.0
		ketamine hydrochloride	11.0	20.0
	9:30:00	**diltiazem hydrochloride**	9.0	13.0
		docetaxel injection	27.0	20.0
		ketamine hydrochloride	12.0	21.0

108 rows × 2 columns

Notice that a **multi-index** dataframe has been created.

We can use `reset_index()` to remove the multi-index.

In [21]:
```
data_tidy = data_tidy.reset_index()
data_tidy
```

Out[21]:

Parameter	Date	time	Drug_Name	Pressure	Temperature
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0
...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0

108 rows × 5 columns

We can rename our `index` column from `Parameter` to simply `None`.

In [22]:
```
data_tidy.columns.name = None
data_tidy.head()
```

Out[22]:

	Date	time	Drug_Name	Pressure	Temperature
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0

Pivot Table

Now suppose we want to find some insights, like **mean temperature day-wise**.

Can we use pivot to find the day-wise mean value of temperature for each drug?

In [23]:
```
data_tidy.pivot(index=['Drug_Name'],
                columns = 'Date',
                values=['Temperature'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[23], line 1
----> 1 data_tidy.pivot(index=['Drug_Name'],
      2                 columns = 'Date',
      3                 values=['Temperature'])

File ~/anaconda3/lib/python3.11/site-packages/pandas/util/_decorators.py:33
1, in deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*ar
gs, **kwargs)
    325 if len(args) > num_allow_args:
    326     warnings.warn(
    327         msg.format(arguments=_format_argument_list(allow_args)),
    328         FutureWarning,
    329         stacklevel=find_stack_level(),
    330     )
--> 331 return func(*args, **kwargs)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/frame.py:8567, in
DataFrame.pivot(self, index, columns, values)
   8561 @Substitution("")
   8562 @Appender(_shared_docs["pivot"])
   8563 @deprecate_nonkeyword_arguments(version=None, allowed_args=["sel
f"])
   8564 def pivot(self, index=None, columns=None, values=None) -> DataFram
e:
   8565     from pandas.core.reshape.pivot import pivot
-> 8567     return pivot(self, index=index, columns=columns, values=values)

File ~/anaconda3/lib/python3.11/site-packages/pandas/util/_decorators.py:33
1, in deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*ar
gs, **kwargs)
    325 if len(args) > num_allow_args:
    326     warnings.warn(
    327         msg.format(arguments=_format_argument_list(allow_args)),
    328         FutureWarning,
    329         stacklevel=find_stack_level(),
    330     )
--> 331 return func(*args, **kwargs)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/reshape/pivot.py:
540, in pivot(data, index, columns, values)
    536         indexed = data._constructor_sliced(data[values]._values, in
dex=multiindex)
    537 # error: Argument 1 to "unstack" of "DataFrame" has incompatible ty
pe "Union
    538 # [List[Any], ExtensionArray, ndarray[Any, Any], Index, Series]"; e
xpected
    539 # "Hashable"
--> 540 return indexed.unstack(columns_listlike)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/frame.py:9112, in
DataFrame.unstack(self, level, fill_value)
   9050 """
   9051 Pivot a level of the (necessarily hierarchical) index labels.
   9052
   (...)
   9108 dtype: float64
   9109 """
   9110 from pandas.core.reshape.reshape import unstack
-> 9112 result = unstack(self, level, fill_value)
   9114 return result.__finalize__(self, method="unstack")

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.p
```

```
y:476, in unstack(obj, level, fill_value)
    474 if isinstance(obj, DataFrame):
    475     if isinstance(obj.index, MultiIndex):
--> 476         return _unstack_frame(obj, level, fill_value=fill_value)
    477     else:
    478         return obj.T.stack(dropna=False)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.p
y:499, in _unstack_frame(obj, level, fill_value)
    497 def _unstack_frame(obj: DataFrame, level, fill_value=None):
    498     assert isinstance(obj.index, MultiIndex)  # checked by caller
--> 499     unstacker = _Unstacker(obj.index, level=level, constructor=obj.
_constructor)
    501     if not obj._can_fast_transpose:
    502         mgr = obj._mgr.unstack(unstacker, fill_value=fill_value)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.p
y:137, in _Unstacker.__init__(self, index, level, constructor)
    129 if num_cells > np.iinfo(np.int32).max:
    130     warnings.warn(
    131         f"The following operation may generate {num_cells} cells "
    132         f"in the resulting pandas object.",
    133         PerformanceWarning,
    134         stacklevel=find_stack_level(),
    135     )
--> 137 self._make_selectors()

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/reshape/reshape.p
y:189, in _Unstacker._make_selectors(self)
    186 mask.put(selector, True)
    188 if mask.sum() < len(self.index):
--> 189     raise ValueError("Index contains duplicate entries, cannot resh
ape")
    191 self.group_index = comp_index
    192 self.mask = mask

ValueError: Index contains duplicate entries, cannot reshape
```

Why did we get an error?

- We need to find the **average** of temperature values throughout a day.
- If you notice, the error shows **duplicate entries**.

Hence, the index values should be unique entry for each row.

What can we do to get our required mean values then?

In [24]:
```
pd.pivot_table(data_tidy, index='Drug_Name', columns='Date', values=['Tempe
```

Out[24]:

	Temperature		
Date	**15-10-2020**	**16-10-2020**	**17-10-2020**
Drug_Name			
diltiazem hydrochloride	21.454545	37.454545	15.636364
docetaxel injection	20.750000	51.454545	17.500000
ketamine hydrochloride	23.555556	11.500000	18.500000

This function is similar to `pivot()`, with an extra feature of an aggregator.

How does `pivot_table()` work?

- The initial parameters are same as what we use in `pivot()` .
- As an extra parameter, we pass the **type of aggregator**.

Note:

- We could have done this using `groupby` too.
- In fact, `pivot_table` uses `groupby` in the backend to group the data and perform the aggregration.
- The only difference is in the type of output we get using both the functions.

Similarly, what if we want to find the minimum values of temperature and pressure on a particular date?

In [25]:
```
pd.pivot_table(data_tidy, index='Drug_Name', columns='Date', values=['Tempe
```

Out[25]:

	Pressure			Temperature		
Date	**15-10-2020**	**16-10-2020**	**17-10-2020**	**15-10-2020**	**16-10-2020**	**17-10-2020**
Drug_Name						
diltiazem hydrochloride	11.0	18.0	3.0	20.0	34.0	10.0
docetaxel injection	22.0	23.0	20.0	17.0	46.0	12.0
ketamine hydrochloride	7.0	12.0	8.0	20.0	8.0	13.0

Binning

Sometimes, we would want our data to be in **categorical** form instead of **continuous/numerical**.

- Let's say, instead of knowing specific test values of a month, I want to know its type.
- Depending on the level of granularity, we want to have - Low, Medium, High, Very High.

How can we derive bins/buckets from continous data?

- use `pd.cut()`

Let's try to use this on our `Temperature` column to categorise the data into bins.

But to define categories, let's first check `min` and `max` temperature values.

In [26]:
```
data_tidy
```

Out[26]:

	Date	time	Drug_Name	Pressure	Temperature
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0
...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0

108 rows × 5 columns

In [27]:

```
print(data_tidy['Temperature'].min(), data_tidy['Temperature'].max())
```

```
8.0 58.0
```

Here,

- Min value = 8
- Max value = 58

Lets's keep some buffer for future values and take the range from 5-60 (instead of 8-58).

We'll divide this data into **4 bins** of 10-15 values each.

In [28]:

```
temp_points = [5, 20, 35, 50, 60]

temp_labels = ['low','medium','high','very_high'] # labels define the sever
```

In [29]:

```
data_tidy['temp_cat'] = pd.cut(data_tidy['Temperature'], bins=temp_points,
data_tidy.head()
```

Out[29]:

	Date	time	Drug_Name	Pressure	Temperature	temp_cat
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	low
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	medium
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	medium
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	low
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	medium

In [30]:

```
data_tidy['temp_cat'].value_counts()
```

Out[30]:
```
low          45
medium       30
high         15
very_high     5
Name: temp_cat, dtype: int64
```

Note: By default, `pd.cut()` creates intervals of the form (x, y] — which includes the right endpoint but excludes the left one.

In [ ]:
```

```

Pandas 5

Content

- Null/Missing values
  - `None` vs `NaN` values
  - `isna()` & `isnull()`
- Removing null values
  - `dropna()`
- Data Imputation
  - `fillna()`
- String methods
- Datetime values
- Writing to a file

In [2]:
```
import pandas as pd
import numpy as np

data = pd.read_csv('Pfizer_1.csv')

data_melt = pd.melt(data,id_vars = ['Date', 'Drug_Name', 'Parameter'],
            var_name = "time",
            value_name = 'reading')

data_tidy = data_melt.pivot(index=['Date','time', 'Drug_Name'],
                                   columns = 'Parameter',
                                   values='reading')

data_tidy = data_tidy.reset_index()
data_tidy.columns.name = None
```

In [3]:
```
data.head()
```

Out[3]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	NaN	21.0	21.0	22
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	NaN	11.0	13.0	14
2	15-10-2020	docetaxel injection	Temperature	NaN	17.0	18.0	NaN	17.0	18
3	15-10-2020	docetaxel injection	Pressure	NaN	22.0	22.0	NaN	22.0	23
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	NaN	NaN	27.0	NaN	26

In [4]:
```
data_melt.head()
```

Out[4]:

	Date	Drug_Name	Parameter	time	reading
0	15-10-2020	diltiazem hydrochloride	Temperature	1:30:00	23.0
1	15-10-2020	diltiazem hydrochloride	Pressure	1:30:00	12.0
2	15-10-2020	docetaxel injection	Temperature	1:30:00	NaN
3	15-10-2020	docetaxel injection	Pressure	1:30:00	NaN
4	15-10-2020	ketamine hydrochloride	Temperature	1:30:00	24.0

In [5]:
```
data_tidy.head()
```

Out[5]:

	Date	time	Drug_Name	Pressure	Temperature
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0

`None` vs `NaN`

If you notice, there are many `NaN` values in our data.

What are these `NaN` values?

- They are basically **missing/null values**.
- A null value signifies an **empty cell/no data**.

There can be 2 kinds of missing values:

1. `None`
2. `NaN` (Not a Number)

Whats the difference between the `None` and `NaN`?

Both `None` and `NaN` can be used for missing values, but their representation and behaviour may differ based on the **column's data type**.

In [6]:
```
type(None)
```

Out[6]:
```
NoneType
```

In [7]:
```
type(np.nan)
```

Out[7]:
```
float
```

1. **None in Non-numeric** columns: None can be used directly, and it will appear as None.

2. **None in Numeric** columns: Pandas automatically converts None to NaN.
3. **NaN in Numeric** columns: NaN is used to represent missing values and appears as NaN.
4. **NaN in Non-numeric** Columns: NaN can be used, and it appears as NaN.

In [8]:
```
pd.Series([1, np.nan, 2, None])
```

Out[8]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For **numerical** type, Pandas changes `None` to `NaN`.

In [9]:
```
pd.Series(["1", "np.nan", "2", None])
```

Out[9]:
```
0         1
1    np.nan
2         2
3      None
dtype: object
```

For **object** type, the `None` is preserved and not changed to `NaN`.

`isna()` & `isnull()`

How to get the count of missing values for each row/column?

- `df.isna()`
- `df.isnull()`

In [10]:
```
data.isna().head()
```

Out[10]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7
0	False	False	False	False	False	True	False	False	False	
1	False	False	False	False	False	True	False	False	False	
2	False	False	False	True	False	False	True	False	False	
3	False	False	False	True	False	False	True	False	False	
4	False	False	False	False	True	True	False	True	False	

In [11]:
```
data.isnull().head()
```

Out[11]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7
0	False	False	False	False	False	True	False	False	False	
1	False	False	False	False	False	True	False	False	False	
2	False	False	False	True	False	False	True	False	False	
3	False	False	False	True	False	False	True	False	False	
4	False	False	False	False	True	True	False	True	False	

Notice that both `isna()` and `isnull()` give the same results.

But why do we have two methods, `isna()` and `isnull()` for the same operation?

- `isnull()` is just an alias for `isna()`

In [12]:

```
pd.isnull
```

Out[12]:

```
<function pandas.core.dtypes.missing.isna(obj: 'object') -> 'bool | npt.NDArray[np.bool_] | NDFrame'>
```

In [13]:

```
pd.isna
```

Out[13]:

```
<function pandas.core.dtypes.missing.isna(obj: 'object') -> 'bool | npt.NDArray[np.bool_] | NDFrame'>
```

As we can see, the function signature is same for both.

- `isna()` returns a **boolean dataframe**, with each cell as a boolean value.
- This value corresponds to **whether the cell has a missing value**.
- On top of this, we can use `.sum()` to find the count of the missing values.

In [14]:

```
data.isna().sum()
```

Out[14]:

```
Date         0
Drug_Name    0
Parameter    0
1:30:00      2
2:30:00      2
3:30:00      6
4:30:00      4
5:30:00      2
6:30:00      0
7:30:00      2
8:30:00      4
9:30:00      2
10:30:00     0
11:30:00     2
12:30:00     0
dtype: int64
```

This gives us the total number of missing values in each column.

How can we get the number of missing values in each row?

In [15]:

```
data.isna().sum(axis=1)
```

Out[15]:
```
0     1
1     1
2     4
3     4
4     3
5     3
6     1
7     1
8     1
9     1
10    2
11    2
12    1
13    1
14    0
15    0
16    0
17    0
dtype: int64
```

Note: By default, the value is `axis=0` for `sum()`.

We now have identified the null count, but how do we deal with them?

We have two options:

- Delete the rows/columns containing the null values.
- Fill the missing values with some data/estimate.

Let's first look at deleting the rows.

Removing null values

How can we drop rows containing null values?

In [16]:
```
data.dropna()
```

Out[16]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
14	17-10-2020	docetaxel injection	Temperature	12.0	13.0	14.0	15.0	16.0	17
15	17-10-2020	docetaxel injection	Pressure	20.0	22.0	22.0	22.0	22.0	23
16	17-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	16.0	17.0	18
17	17-10-2020	ketamine hydrochloride	Pressure	8.0	9.0	10.0	11.0	11.0	12

Notice that rows with even a single missing value have been deleted.

What if we want to delete the columns having missing value?

In [18]:
```
data.dropna(axis=1)
```

Out[18]:

	Date	Drug_Name	Parameter	6:30:00	10:30:00	12:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	22	20	21
1	15-10-2020	diltiazem hydrochloride	Pressure	14	18	20
2	15-10-2020	docetaxel injection	Temperature	18	23	25
3	15-10-2020	docetaxel injection	Pressure	23	26	28
4	15-10-2020	ketamine hydrochloride	Temperature	26	22	20
5	15-10-2020	ketamine hydrochloride	Pressure	9	9	11
6	16-10-2020	diltiazem hydrochloride	Temperature	38	40	42
7	16-10-2020	diltiazem hydrochloride	Pressure	23	24	27
8	16-10-2020	docetaxel injection	Temperature	49	56	58
9	16-10-2020	docetaxel injection	Pressure	27	28	30
10	16-10-2020	ketamine hydrochloride	Temperature	12	13	15
11	16-10-2020	ketamine hydrochloride	Pressure	15	16	18
12	17-10-2020	diltiazem hydrochloride	Temperature	16	14	10
13	17-10-2020	diltiazem hydrochloride	Pressure	8	11	14
14	17-10-2020	docetaxel injection	Temperature	17	21	23
15	17-10-2020	docetaxel injection	Pressure	23	28	28
16	17-10-2020	ketamine hydrochloride	Temperature	18	22	24
17	17-10-2020	ketamine hydrochloride	Pressure	12	13	15

Notice that every column which had even a single missing value has been deleted.

But what are the problems with deleting rows/columns?

- loss of valuable data

So instead of dropping, it would be better to **fill the missing values with some data**.

Data Imputation

How can we fill the missing values with some data?

In [19]:
```
data.fillna(0).head()
```

Out[19]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	0.0	21.0	21.0	22
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	0.0	11.0	13.0	14
2	15-10-2020	docetaxel injection	Temperature	0.0	17.0	18.0	0.0	17.0	18
3	15-10-2020	docetaxel injection	Pressure	0.0	22.0	22.0	0.0	22.0	23
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	0.0	0.0	27.0	0.0	26

What is `fillna(0)` doing?

- It fills all the missing values with 0.

We can do the same on a particular column too.

In [20]:

```
data['2:30:00'].fillna(0)
```

Out[20]:

```
0     22.0
1     13.0
2     17.0
3     22.0
4      0.0
5      0.0
6     35.0
7     19.0
8     47.0
9     24.0
10     9.0
11    12.0
12    19.0
13     4.0
14    13.0
15    22.0
16    14.0
17     9.0
Name: 2:30:00, dtype: float64
```

Note:

Handling missing value completely depends on the business problem.

However, in general practice (assuming you have a large dataset) -

- if the missing values are minimal (\<5% of rows), dropping them is acceptable.
- for substantial missing values (>10% of rows), use a suitable imputation technique.
- if a column has over 50% of null values, drop that column (unless it's very crucial for the analysis).

What other values can we use to fill the missing values?

We can use some kind of estimator too.

- mean (average value)
- median
- mode (most frequently occuring value)

How would you calculate the mean of the column `2:30:00` ?

In [21]:
```
data['2:30:00'].mean()
```

Out[21]:
```
18.8125
```

Now let's fill the `NaN` values with the mean value of the column.

In [22]:
```
data['2:30:00'].fillna(data['2:30:00'].mean())
```

Out[22]:
```
0     22.0000
1     13.0000
2     17.0000
3     22.0000
4     18.8125
5     18.8125
6     35.0000
7     19.0000
8     47.0000
9     24.0000
10     9.0000
11    12.0000
12    19.0000
13     4.0000
14    13.0000
15    22.0000
16    14.0000
17     9.0000
Name: 2:30:00, dtype: float64
```

But this doesn't feel right. What could be wrong with this?

Can we use the mean of all compounds as average for our estimator?

- Different drugs have different characteristics.
- We can't simply do an average and fill the null values.

Then what could be the solution here?

We could fill the null values of respective compounds with their respective means.

How can we form a column with mean temperature of respective compounds?

- We can use `apply()`

Let's first create a function to calculate the mean.

In [23]:
```
def temp_mean(x):
  x['Temperature_avg'] = x['Temperature'].mean()
  return x
```

Now we can form a new column based on the average values of temperature for each drug.

In [25]:
```
data_tidy = data_tidy.groupby(["Drug_Name"]).apply(temp_mean)
data_tidy
```

```
/var/folders/zk/yt14z40j2lb2lz548fqr3v9m0000gn/T/ipykernel_59236/264220330
0.py:1: FutureWarning: Not prepending group keys to the result index of tra
nsform-like apply. In the future, the group keys will be included in the in
dex, regardless of whether the applied function returns a like-indexed obje
ct.
To preserve the previous behavior, use

        >>> .groupby(..., group_keys=False)

To adopt the future behavior and silence this warning, use

        >>> .groupby(..., group_keys=True)
  data_tidy = data_tidy.groupby(["Drug_Name"]).apply(temp_mean)
```

Out[25]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097
...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677

108 rows × 6 columns

In [26]:
```
data_tidy = data_tidy.groupby(["Drug_Name"],group_keys=False).apply(temp_mea
data_tidy
```

Out[26]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097
...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677

108 rows × 6 columns

Now we fill the null values in `Temperature` using this new column.

In [27]:

```
data_tidy['Temperature'].fillna(data_tidy["Temperature_avg"], inplace=True)
data_tidy
```

Out[27]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097
...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677

108 rows × 6 columns

In [28]:

```
data_tidy.isna().sum()
```

Out[28]:

```
Date               0
time               0
Drug_Name          0
Pressure          13
Temperature        0
Temperature_avg    0
dtype: int64
```

Great!

We have removed the null values from our `Temperature` column.

Let's do the same for `Pressure`.

In [29]:

```
def pr_mean(x):
  x['Pressure_avg'] = x['Pressure'].mean()
  return x
data_tidy=data_tidy.groupby(["Drug_Name"]).apply(pr_mean)
data_tidy['Pressure'].fillna(data_tidy["Pressure_avg"], inplace=True)
data_tidy
```

```
/var/folders/zk/yt14z40j2lb2lz548fqr3v9m0000gn/T/ipykernel_59236/258637458
5.py:4: FutureWarning: Not prepending group keys to the result index of tra
nsform-like apply. In the future, the group keys will be included in the in
dex, regardless of whether the applied function returns a like-indexed obje
ct.
To preserve the previous behavior, use

	>>> .groupby(..., group_keys=False)

To adopt the future behavior and silence this warning, use

	>>> .groupby(..., group_keys=True)
  data_tidy=data_tidy.groupby(["Drug_Name"]).apply(pr_mean)
```

Out[29]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871
...	...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097	25.483871
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677	11.935484
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485	15.424242
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097	25.483871
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677	11.935484

108 rows × 7 columns

In [30]:
```
data_tidy.isna().sum()
```

Out[30]:
```
Date               0
time               0
Drug_Name          0
Pressure           0
Temperature        0
Temperature_avg    0
Pressure_avg       0
dtype: int64
```

How to decide if we should impute the missing values with `mean`, `median` or `mode`?

1. `Mean`: Use when dealing with numerical data that is normally distributed and not heavily skewed by outliers.

2. `Median`: Preferable when data is skewed or contains outliers. It's suitable for ordinal or interval data.

3. `Mode`: Suitable for categorical or nominal data where there are distinct categories.

Question

Based on the given DataFrame, which of the following statements regarding data imputation is mostly accurate?

```
|   CustomerID   |  TransactionAmount  |     Gender     |
Age  |  ProductCategory  |
|----------------|---------------------|----------------|----
---|-------------------|
|       101      |         20          |     Male       |
35  |       Apparel     |
|       102      |        NaN          |     Female     |
28  |       NaN         |
|       103      |         15          |     Female     |
NaN  |       Electronics |
|       104      |         30          |     NaN        |
42  |       Electronics |
|       105      |        150          |     Male       |
30  |       Apparel     |

A) Imputing missing values in the "TransactionAmount" column
using the mean of the available values may not be suitable
due to potential skewness caused by outliers.
B) Imputing missing values in the "TransactionAmount" column
using the median of the available values may be suitable to
handle skewness due to outliers.
C) The presence of missing values in the "Gender" column can
be effectively handled by imputing the most frequent category
(mode).
D) All of the above
```

Answer: All of the above

Explanation:

- Option A is correct because imputing missing values in the "TransactionAmount" column with the mean may not be appropriate if the data contains outliers. Outliers can significantly skew the mean, leading to inaccurate imputations.
- Option B is correct because as the data is skewed, the median that is roubst to outliers can better impute the missing data

- Option C is correct because for the "Gender" categorical column, the most frequently occuring category can be used to impute as gender is unlikely to exhibit significant variation in a dataset of customer transactions.

String methods

What kind of questions can we use string methods for?

- Find rows which contains a particular string.

Say,

How you can you filter rows containing "hydrochloride" in their drug name?

In [31]:
```
data_tidy.loc[data_tidy['Drug_Name'].str.contains('hydrochloride')].head()
```

Out[31]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242
5	15-10-2020	11:30:00	ketamine hydrochloride	9.0	21.0	17.709677	11.935484
6	15-10-2020	12:30:00	diltiazem hydrochloride	20.0	21.0	24.848485	15.424242

- So in general, we will be using the following format: `Series.str.function()`

- `Series.str` can be used to access the values of the series as strings and apply several methods to it.

Now suppose we want to form a new column based on the year of the experiments?

What can we do form a column containing the year?

In [32]:
```
data_tidy['Date'].str.split('-')
```

Out[32]:
```
0      [15, 10, 2020]
1      [15, 10, 2020]
2      [15, 10, 2020]
3      [15, 10, 2020]
4      [15, 10, 2020]
            ...
103    [17, 10, 2020]
104    [17, 10, 2020]
105    [17, 10, 2020]
106    [17, 10, 2020]
107    [17, 10, 2020]
Name: Date, Length: 108, dtype: object
```

To extract the year, we need to select the last element of each list.

In [33]:
```
data_tidy['Date'].str.split('-').apply(lambda x:x[2])
```

Out[33]:
```
0      2020
1      2020
2      2020
3      2020
4      2020
       ...
103    2020
104    2020
105    2020
106    2020
107    2020
Name: Date, Length: 108, dtype: object
```

But there are certain problems with this approach.

- The **dtype of the output is still an object**, we would prefer a number type.
- The date format will always **not be in day-month-year**, it can vary.

Thus, to work with such date-time type of data, we can use a special method from Pandas.

Datetime

How can we handle datetime data types?

- We can use the `to_datetime()` function of Pandas
- It takes as input:
  - Array/Scalars with values having proper date/time format
  - `dayfirst` : Indicating if the day comes first in the date format used
  - `yearfirst` : Indicates if year comes first in the date format used

Let's first merge our `Date` and `Time` columns into a new `timestamp` column.

In [34]:
```
data_tidy['timestamp'] = data_tidy['Date'] + " " + data_tidy['time']
```

In [35]:
```
data_tidy.head()
```

Out[35]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg	ti
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242	
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871	
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484	
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242	
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871	

In [36]:

```
data_tidy['timestamp'] = pd.to_datetime(data_tidy['timestamp'])
data_tidy
```

Out[36]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871
...	...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097	25.483871
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677	11.935484
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485	15.424242
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097	25.483871
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677	11.935484

108 rows × 8 columns

In [37]:

```
data_tidy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 108 entries, 0 to 107
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Date             108 non-null    object
 1   time             108 non-null    object
 2   Drug_Name        108 non-null    object
 3   Pressure         108 non-null    float64
 4   Temperature      108 non-null    float64
 5   Temperature_avg  108 non-null    float64
 6   Pressure_avg     108 non-null    float64
 7   timestamp        108 non-null    datetime64[ns]
dtypes: datetime64[ns](1), float64(4), object(3)
memory usage: 11.7+ KB
```

The type of `timestamp` column has been changed from `object` to `datetime`.

Now, let's look at a single timestamp using Pandas.

How can we extract information from a single timestamp using Pandas?

In [38]:
```
ts = data_tidy['timestamp'][0]
ts
```

Out[38]:
```
Timestamp('2020-10-15 10:30:00')
```

In [39]:
```
ts.year, ts.month, ts.day, ts.month_name()
```

Out[39]:
```
(2020, 10, 15, 'October')
```

In [40]:
```
ts.hour, ts.minute, ts.second
```

Out[40]:
```
(10, 30, 0)
```

This data parsing from `string` to `datetime` makes it easier to work with such data.

We can use this data from the columns as a whole using `.dt` object.

In [41]:
```
data_tidy['timestamp'].dt
```

Out[41]:
```
<pandas.core.indexes.accessors.DatetimeProperties object at 0x12206b950>
```

- `dt` gives properties of values in a column.
- From this `DatetimeProperties` of column `'end'`, we can extract `year`.

In [42]:
```
data_tidy['timestamp'].dt.year
```

Out[42]:
```
0      2020
1      2020
2      2020
3      2020
4      2020
       ... 
103    2020
104    2020
105    2020
106    2020
107    2020
Name: timestamp, Length: 108, dtype: int64
```

We can use `strfttime` (**short for stringformat time**), to modify our datetime format.

Let's learn this with the help of few examples.

In [43]:
```
data_tidy['timestamp'][0]
```

Out[43]:
```
Timestamp('2020-10-15 10:30:00')
```

In [44]:
```
print(data_tidy['timestamp'][0].strftime('%Y')) # formatter for year
```

```
2020
```

Similarly we can combine the format types to modify the datetime format as per our convinience.

A comprehensive list of other formats can be found here:
https://pandas.pydata.org/docs/reference/api/pandas.Period.strftime.html

In [45]:
```
data_tidy['timestamp'][0].strftime('%m-%d')
```

Out[45]:
```
'10-15'
```

Writing to a file

How can we write our dataframe to a CSV file?

- We have to provide the `path` and `file_name` in which we want to store the data.

In [46]:
```
data_tidy.to_csv('pfizer_tidy.csv', sep=",", index=False)
```

Setting `index=False` will not inlcude the index column while writing.

In [ ]:
```

```