PANDAS

Complete Revision Notes

RAHUL KUMAR

https://www.linkedin.com/in/rahul-kumar-1212a6141/

Outline

- **Installation of pandas** - Importing pandas - Importing the dataset - Dataframe/Series - **Basic ops on a DataFrame** - df.info() - df.head() - df.tail() - df.shape() - **Creating Dataframe from Scratch** - **Basic ops on columns** - Different ways of accessing cols - Check for Unique values - Rename column - Deleting col - Creating new cols - **Basic ops on rows** - Implicit/explicit index - Quiz 2 - df.index - Indexing in series - Slicing in series - loc/iloc - Indexing/Slicing in dataframe - Adding a row - Deleting a row - **Working with both rows and columns** - **More in-built ops in pandas** - sum() - count() - mean() - **Sorting** - **Concatenation** - pd.concat() - axis for concat - **Merge** - Concat v/s Merge - `left_on` and `right_on` - Joins - **Intoduction to IMDB dataset** - Reading two datasets - **Merging the dataframes** - `unique()` and `nunique()` - `isin()` - Using Left Join for `merge()` - **Feature Exploration** - Create new features - **Fetching data using pandas** - Quering from dataframe - Masking, Filtering, `&` and `|` - ASSESSMENT: - **Apply** - **Grouping** - Split, Apply, Combine - `groupby()` - **Group based Aggregates** Coding (E) - **Group based Filtering** - **Group based Apply** - `apply()` - **Restructuring data** - pd.melt() - pd.pivot() - pd.pivot_table() - pd.cut() - Dealing with Missing Values - None and nan values - isna() and isnull() - String method in pandas - Handling datetime - **Writing to a file**

Importing Pandas

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

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

Introduction: Why to use Pandas?

How is it different from numpy ?

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

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

Pandas can work with numbers and strings together

So lets see how we can use pandas

Imagine that you are a Data Scientist with McKinsey

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

- We have to analyse the data and draw **inferences** meaningful to the company

Reading dataset in Pandas

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

In [3]:
```
df = pd.read_csv(r"C:\Users\kumar\Downloads\mckinsey.csv")
```

Now how should we read this dataset?

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

In [4]:
```
df
```

Out[4]:

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

1704 rows × 6 columns

Dataframe and Series

What can we observe from the above dataset ?

We can see that it has:

- 6 columns
- 1704 rows

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

In [5]:
```
type(df)
```

Out[5]:
```
pandas.core.frame.DataFrame
```

Its a **pandas DataFrame**

What is a pandas DataFrame ?

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

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

In [6]:
```
df["country"]
```

Out[6]:
```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

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

Now what is the data-type of a column?

In [7]:
```
type(df["country"])
```

Out[7]:
```
pandas.core.series.Series
```

Its a **pandas Series**

What is a pandas Series ?

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

What exactly does that mean?

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

Now we have understood what Series and DataFrames are

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

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

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   country     1704 non-null   object
 1   year        1704 non-null   int64 
```

```
 2   population  1704 non-null   int64
 3   continent   1704 non-null   object
 4   life_exp    1704 non-null   float64
 5   gdp_cap     1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
```

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

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

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

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

In [9]:
```
df.head()
```

Out[9]:

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

It **Prints top 5 rows by default**

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

In [10]:
```
df.head(20)
```

Out[10]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1977	14880372	Asia	38.438	786.113360
6	Afghanistan	1982	12881816	Asia	39.854	978.011439
7	Afghanistan	1987	13867957	Asia	40.822	852.395945
8	Afghanistan	1992	16317921	Asia	41.674	649.341395
9	Afghanistan	1997	22227415	Asia	41.763	635.341351
10	Afghanistan	2002	25268405	Asia	42.129	726.734055
11	Afghanistan	2007	31889923	Asia	43.828	974.580338
12	Albania	1952	1282697	Europe	55.230	1601.056136
13	Albania	1957	1476505	Europe	59.280	1942.284244

14	Albania	1962	1728137	Europe	64.820	2312.888958
15	Albania	1967	1984060	Europe	66.220	2760.196931
16	Albania	1972	2263554	Europe	67.690	3313.422188
17	Albania	1977	2509048	Europe	68.930	3533.003910
18	Albania	1982	2780097	Europe	70.420	3630.880722
19	Albania	1987	3075321	Europe	72.000	3738.932735

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

In [11]:
```
df.tail(20) #Similar to head
```

Out[11]:

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

How can we find the shape of the dataframe?

In [12]:
```
df.shape
```

Out[12]:
```
(1704, 6)
```

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

Now we know how to do some basic operations on dataframes

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

Let's take a subset of the original dataset

In [13]:
```
df.head(3) # We take the first 3 rows to create our dataframe
```

Out[13]:

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

How can we create a DataFrame from scratch?

Approach 1: Row-oriented

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

In [14]:
```
pd.DataFrame([['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ],
              ['Afghanistan',1957, 9240934, 'Asia', 30.332, 820.853030 ],
              ['Afghanistan',1962, 102267083, 'Asia', 31.997, 853.100710 ]],
             columns=['country','year','population','continent','life_exp','gdp_cap'])
```

Out[14]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	102267083	Asia	31.997	853.100710

Can you create a single row dataframe?

In [15]:
```
pd.DataFrame(['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ],
             columns=['country','year','population','continent','life_exp','gdp_cap'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [15], in <cell line: 1>()
----> 1 pd.DataFrame(['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ],
      2              columns=['country','year','population','continent','life_exp','gdp_
cap'])

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:737, in DataFrame.__init__(self,
 data, index, columns, dtype, copy)
    729         mgr = arrays_to_mgr(
    730             arrays,
    731             columns,
   (...)
    734             typ=manager,
    735         )
```

```
    736     else:
--> 737         mgr = ndarray_to_mgr(
    738             data,
    739             index,
    740             columns,
    741             dtype=dtype,
    742             copy=copy,
    743             typ=manager,
    744         )
    745 else:
    746     mgr = dict_to_mgr(
    747         {},
    748         index,
   (...)
    751         typ=manager,
    752     )

File ~\anaconda3\lib\site-packages\pandas\core\internals\construction.py:351, in ndarray_to_mgr(values, index, columns, dtype, copy, typ)
    346 # _prep_ndarray ensures that values.ndim == 2 at this point
    347 index, columns = _get_axes(
    348     values.shape[0], values.shape[1], index=index, columns=columns
    349 )
--> 351 _check_values_indices_shape_match(values, index, columns)
    353 if typ == "array":
    355     if issubclass(values.dtype.type, str):

File ~\anaconda3\lib\site-packages\pandas\core\internals\construction.py:422, in _check_values_indices_shape_match(values, index, columns)
    420 passed = values.shape
    421 implied = (len(index), len(columns))
--> 422 raise ValueError(f"Shape of passed values is {passed}, indices imply {implied}")

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

Why did this give an error?

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

In [16]:
```
pd.DataFrame([['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ]],
             columns=['country','year','population','continent','life_exp','gdp_cap'])
```

Out[16]:

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

Approach 2: Column-oriented

In [17]:
```
pd.DataFrame({'country':['Afghanistan', 'Afghanistan'], 'year':[1952,1957],
              'population':[842533, 9240934], 'continent':['Asia', 'Asia'],
              'life_exp':[28.801, 30.332], 'gdp_cap':[779.445314, 820.853030]})
```

Out[17]:

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

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

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

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

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

Thinking from database perspective:

- Adding data
- Removing data
- Updating/Modifying data

and so on

Basic operations on columns

Now what operations can we do using columns?

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

and so on.

We can see that our dataset has 6 cols

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

How can we get the names of all these cols ?

We can do it in two ways:

1. df.columns
2. df.keys

In [18]:
```
df.columns  # using attribute `columns` of dataframe
```

Out[18]:
```
Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_cap'], dtype='object')
```

In [19]:
```
df.keys()  # using method keys() of dataframe
```

Out[19]:
```
Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_cap'], dtype='object')
```

Note:

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

In [20]:
```
# df['country'].head()  # Gives values in Top 5 rows pertaining to the key
```

Pandas DataFrame and Series are specialised dictionary

But what is so "special" about this dictionary?

It can take multiple keys

In [21]:
```
df[['country', 'life_exp']].head()
```

Out[21]:

	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 [22]:
```
df[['country']].head()
```

Out[22]:

	country
0	Afghanistan
1	Afghanistan
2	Afghanistan
3	Afghanistan
4	Afghanistan

Note:

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

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

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

How can we find the countries that have been surveyed ?

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

How can we find unique values in a column?

In [23]:
```
df['country'].unique()
```

Out[23]:
```
array(['Afghanistan', 'Albania', 'Algeria', 'Angola', 'Argentina',
       'Australia', 'Austria', 'Bahrain', 'Bangladesh', 'Belgium',
       'Benin', 'Bolivia', 'Bosnia and Herzegovina', 'Botswana', 'Brazil',
       'Bulgaria', 'Burkina Faso', 'Burundi', 'Cambodia', 'Cameroon',
       'Canada', 'Central African Republic', 'Chad', 'Chile', 'China',
       'Colombia', 'Comoros', 'Congo, Dem. Rep.', 'Congo, Rep.',
       'Costa Rica', "Cote d'Ivoire", 'Croatia', 'Cuba', 'Czech Republic',
       'Denmark', 'Djibouti', 'Dominican Republic', 'Ecuador', 'Egypt',
       'El Salvador', 'Equatorial Guinea', 'Eritrea', 'Ethiopia',
       'Finland', 'France', 'Gabon', 'Gambia', 'Germany', 'Ghana',
       'Greece', 'Guatemala', 'Guinea', 'Guinea-Bissau', 'Haiti',
       'Honduras', 'Hong Kong, China', 'Hungary', 'Iceland', 'India',
```

```
'Indonesia', 'Iran', 'Iraq', 'Ireland', 'Israel', 'Italy',
'Jamaica', 'Japan', 'Jordan', 'Kenya', 'Korea, Dem. Rep.',
'Korea, Rep.', 'Kuwait', 'Lebanon', 'Lesotho', 'Liberia', 'Libya',
'Madagascar', 'Malawi', 'Malaysia', 'Mali', 'Mauritania',
'Mauritius', 'Mexico', 'Mongolia', 'Montenegro', 'Morocco',
'Mozambique', 'Myanmar', 'Namibia', 'Nepal', 'Netherlands',
'New Zealand', 'Nicaragua', 'Niger', 'Nigeria', 'Norway', 'Oman',
'Pakistan', 'Panama', 'Paraguay', 'Peru', 'Philippines', 'Poland',
'Portugal', 'Puerto Rico', 'Reunion', 'Romania', 'Rwanda',
'Sao Tome and Principe', 'Saudi Arabia', 'Senegal', 'Serbia',
'Sierra Leone', 'Singapore', 'Slovak Republic', 'Slovenia',
'Somalia', 'South Africa', 'Spain', 'Sri Lanka', 'Sudan',
'Swaziland', 'Sweden', 'Switzerland', 'Syria', 'Taiwan',
'Tanzania', 'Thailand', 'Togo', 'Trinidad and Tobago', 'Tunisia',
'Turkey', 'Uganda', 'United Kingdom', 'United States', 'Uruguay',
'Venezuela', 'Vietnam', 'West Bank and Gaza', 'Yemen, Rep.',
'Zambia', 'Zimbabwe'], dtype=object)
```

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

In [24]:
```
df['country'].value_counts()
```

Out[24]:
```
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 [25]:
```
df.rename({"population": "Population", "country":"Country" }, axis = 1)
```

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

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

- by using the `column` parameter

In [26]:
```
df.rename(columns={"country":"Country"})
```

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

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

In [27]:
```
df.rename({"country": "Country"}, axis = 1, inplace = True)
df
```

Out[27]:

	Country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960

1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

Note

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

Now lets try another way of accessing column vals

In [28]:
```
df.Country
```

Out[28]:
```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: Country, Length: 1704, dtype: object
```

This however doesn't work everytime

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

Problems such as

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

It is generally better to avoid this type of accessing columns

Are all the columns in our data necessary?

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

How can we delete cols in pandas dataframe ?

In [29]:
```
df.drop('continent', axis=1)
```

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

The `drop` function takes two parameters:

- The column name
- The axis

By default the value of `axis` is 0

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

In [30]:
```
df.drop(columns=['continent'])
```

Out[30]:

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

1704 rows × 5 columns

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

Has the column permanently been deleted?

In [31]:
```
df.head()
```

Out[31]:

	Country	year	population	continent	life_exp	gdp_cap

0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106

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

Do you see what's happening here?

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

How can we permanently drop the column?

We can either **re-assign** it

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

  OR

- We can **set parameter `inplace=True`**

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

In [32]:
```
df.drop('continent', axis=1, inplace=True)
```

In [33]:
```
df.head() #we print the head to check
```

Out[33]:

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

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

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

We can either

- use values from **existing columns**

OR

- create our **own values**

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

In [34]:
```
df["year+7"] = df["year"] + 7
df.head()
```

Out[34]:

	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 [35]:
```
df['gdp']=df['gdp_cap'] * df['population']
df.head()
```

Out[35]:

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

As you can see

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

What other operations we can use?

Subtraction, Addition, etc.

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

- We can **create a list**

OR

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

In [36]:
```
df["Own"] = [i for i in range(1704)]  # count of these values should be correct
df
```

Out[36]:

	Country	year	population	life_exp	gdp_cap	year+7	gdp	Own
0	Afghanistan	1952	8425333	28.801	779.445314	1959	6.567086e+09	0
1	Afghanistan	1957	9240934	30.332	820.853030	1964	7.585449e+09	1

2	Afghanistan	1962	10267083	31.997	853.100710	1969	8.758856e+09	2
3	Afghanistan	1967	11537966	34.020	836.197138	1974	9.648014e+09	3
4	Afghanistan	1972	13079460	36.088	739.981106	1979	9.678553e+09	4
...	...	...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	62.351	706.157306	1994	6.508241e+09	1699
1700	Zimbabwe	1992	10704340	60.377	693.420786	1999	7.422612e+09	1700
1701	Zimbabwe	1997	11404948	46.809	792.449960	2004	9.037851e+09	1701
1702	Zimbabwe	2002	11926563	39.989	672.038623	2009	8.015111e+09	1702
1703	Zimbabwe	2007	12311143	43.487	469.709298	2014	5.782658e+09	1703

1704 rows × 8 columns

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

Before that lets drop the newly created cols

In [37]:
```
df.drop(columns=["Own",'gdp', 'year+7'], axis = 1, inplace = True)
df
```

Out[37]:

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

1704 rows × 5 columns

Working with Rows

Just like columns, do rows also have labels?

YES

Notice the indexes in bold against each row

Lets see how can we access these indexes

In [38]:
```
df.index.values
```

Out[38]:
```
array([   0,    1,    2, ..., 1701, 1702, 1703], dtype=int64)
```

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

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

In [39]:
```
df.index = list(range(1, df.shape[0]+1)) # create a list of indexes of same length
df
```

Out[39]:

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

1704 rows × 5 columns

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

Explicit and Implicit Indices

What are these row labels/indices exactly ?

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

Additionally, can series/dataframes can also use python style indexing?

YES

The python style indices are known as **implicit indices**

How can we access explicit index of a particular row?

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

In [40]:
```
df.index[1] #Implicit index 1 gave explicit index 2
```

Out[40]:
```
2
```

But why not use just implicit indexing ?

Explicit indices can be changed to any value of any datatype

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

In [41]:
```
df.index = np.arange(1, df.shape[0]+1, dtype='float')
df
```

Out[41]:

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

1704 rows × 5 columns

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

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

In [42]:
```
sample = df.head()
sample
```

Out[42]:

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

Now what if we want to use string indices?

In [43]:
```
sample.index = ['a', 'b', 'c', 'd', 'e']
sample
```

Out[43]:

	Country	year	population	life_exp	gdp_cap
a	Afghanistan	1952	8425333	28.801	779.445314

b	Afghanistan	1957	9240934	30.332	820.853030
c	Afghanistan	1962	10267083	31.997	853.100710
d	Afghanistan	1967	11537966	34.020	836.197138
e	Afghanistan	1972	13079460	36.088	739.981106

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

Now let's reset our indices back to integers

In [44]:
```
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 [45]:
```
ser = df["Country"]
ser.head(20)
```

Out[45]:
```
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
5     Afghanistan
6     Afghanistan
7     Afghanistan
8     Afghanistan
9     Afghanistan
10    Afghanistan
11    Afghanistan
12    Afghanistan
13        Albania
14        Albania
15        Albania
16        Albania
17        Albania
18        Albania
19        Albania
20        Albania
Name: Country, dtype: object
```

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

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

In [46]:
```
ser[12]
```

Out[46]:
```
'Afghanistan'
```

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

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

Out[47]:
```
6     Afghanistan
7     Afghanistan
8     Afghanistan
9     Afghanistan
```

```
10    Afghanistan
11    Afghanistan
12    Afghanistan
13        Albania
14        Albania
15        Albania
Name: Country, dtype: object
```

This is known as slicing

Notice something different though?

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

Let's try the same for the dataframe now

So how can we access a row in a dataframe?

In [48]:
```
df[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:3621, in Index.get_loc(self, key, method, tolerance)
   3620 try:
-> 3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:

File ~\anaconda3\lib\site-packages\pandas\_libs\index.pyx:136, in pandas._libs.index.IndexEngine.get_loc()

File ~\anaconda3\lib\site-packages\pandas\_libs\index.pyx:163, in pandas._libs.index.IndexEngine.get_loc()

File pandas\_libs\hashtable_class_helper.pxi:5198, in pandas._libs.hashtable.PyObjectHashTable.get_item()

File pandas\_libs\hashtable_class_helper.pxi:5206, in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 0

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

KeyError                                  Traceback (most recent call last)
Input In [48], in <cell line: 1>()
----> 1 df[0]

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:3505, in DataFrame.__getitem__(self, key)
   3503 if self.columns.nlevels > 1:
   3504     return self._getitem_multilevel(key)
-> 3505 indexer = self.columns.get_loc(key)
   3506 if is_integer(indexer):
   3507     indexer = [indexer]

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:3623, in Index.get_loc(self, key, method, tolerance)
   3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:
-> 3623     raise KeyError(key) from err
   3624 except TypeError:
```

```
   3625     # If we have a listlike key, _check_indexing_error will raise
   3626     #  InvalidIndexError. Otherwise we fall through and re-raise
   3627     #  the TypeError.
   3628     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 [ ]:
```
df[5:15]
```

Woah, so the slicing works

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

This can be a cause for confusion

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

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

loc and iloc

1. loc

Allows indexing and slicing that always references the explicit index

In [49]:
```
df.loc[1]
```

Out[49]:
```
Country       Afghanistan
year                 1952
population        8425333
life_exp           28.801
gdp_cap        779.445314
Name: 1, dtype: object
```

In [50]:
```
df.loc[1:3]
```

Out[50]:

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

Did you notice something strange here?

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

2. iloc

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

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

Out[51]:
```
Country       Afghanistan
year                 1957
population        9240934
life_exp           30.332
gdp_cap         820.85303
Name: 2, dtype: object
```

Now will `iloc` also consider the range inclusive?

In [52]:
```
df.iloc[0:2]
```

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

NO

Because **`iloc` works with implicit Python-style indices**

It is important to know about these conceptual differences

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

Which one should we use ?

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

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

For eg: rows 1, 10, 100

In [53]:
```
df.iloc[[1, 10, 100]]
```

Out[53]:

	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

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

What about negative index?

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

In [54]:
```
df.iloc[-1]

# Works and gives last row in dataframe
```

Out[54]:
```
Country          Zimbabwe
year                 2007
population       12311143
life_exp           43.487
gdp_cap         469.709298
Name: 1704, dtype: object
```

In [55]:
```
df.loc[-1]

# Does NOT work
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:3621, in Index.get_loc(self, key, method, tolerance)
   3620 try:
-> 3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:

File ~\anaconda3\lib\site-packages\pandas\_libs\index.pyx:136, in pandas._libs.index.IndexEngine.get_loc()

File ~\anaconda3\lib\site-packages\pandas\_libs\index.pyx:163, in pandas._libs.index.IndexEngine.get_loc()

File pandas\_libs\hashtable_class_helper.pxi:2131, in pandas._libs.hashtable.Int64HashTable.get_item()

File pandas\_libs\hashtable_class_helper.pxi:2140, in pandas._libs.hashtable.Int64HashTable.get_item()

KeyError: -1

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

KeyError                                  Traceback (most recent call last)
Input In [55], in <cell line: 1>()
----> 1 df.loc[-1]

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:967, in _LocationIndexer.__getitem__(self, key)
    964 axis = self.axis or 0
    966 maybe_callable = com.apply_if_callable(key, self.obj)
--> 967 return self._getitem_axis(maybe_callable, axis=axis)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1202, in _LocIndexer._getitem_axis(self, key, axis)
   1200 # fall thru to straight lookup
   1201 self._validate_key(key, axis)
-> 1202 return self._get_label(key, axis=axis)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1153, in _LocIndexer._get_label(self, label, axis)
   1151 def _get_label(self, label, axis: int):
   1152     # GH#5667 this will fail if the label is not present in the axis.
-> 1153     return self.obj.xs(label, axis=axis)

File ~\anaconda3\lib\site-packages\pandas\core\generic.py:3864, in NDFrame.xs(self, key, axis, level, drop_level)
   3862             new_index = index[loc]
   3863 else:
-> 3864     loc = index.get_loc(key)
   3866     if isinstance(loc, np.ndarray):
   3867         if loc.dtype == np.bool_:
```

```
File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:3623, in Index.get_loc(self, key, method, tolerance)
   3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:
-> 3623     raise KeyError(key) from err
   3624 except TypeError:
   3625     # If we have a listlike key, _check_indexing_error will raise
   3626     #  InvalidIndexError. Otherwise we fall through and re-raise
   3627     #  the TypeError.
   3628     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 [56]:
```
temp = df.set_index("Country")
temp
```

Out[56]:

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

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

In [57]:
```
temp.loc['Afghanistan']
```

Out[57]:

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
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`

Now how can we reset our indices back to integers?

In [58]:
```
df.reset_index()
```

Out[58]:

	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 it's creating a new column `index`

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

In [59]:
```
df.reset_index(drop=True) # By using drop=True we can prevent creation of a new column
```

Out[59]:

	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 [60]:
```
df.reset_index(drop=True, inplace=True)
```

Now how can we add a row to our dataframe?

There are multiple ways to do this:

- `append()`
- `loc/iloc`

How can we do add a row using the **append()** method?

In [61]:
```
new_row = {'Country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_
df.append(new_row)
```

```
C:\Users\kumar\AppData\Local\Temp\ipykernel_4240\2797024952.py:2: FutureWarning: The fra
me.append method is deprecated and will be removed from pandas in a future version. Use
pandas.concat instead.
  df.append(new_row)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [61], in <cell line: 2>()
      1 new_row = {'Country': 'India', 'year': 2000,'life_exp':37.08,'population':135000
00,'gdp_cap':900.23}
----> 2 df.append(new_row)

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:9039, in DataFrame.append(self,
 other, ignore_index, verify_integrity, sort)
   8936 """
   8937 Append rows of `other` to the end of caller, returning a new object.
   8938
   (...)
   9029 4  4
   9030 """
   9031 warnings.warn(
   9032     "The frame.append method is deprecated "
   9033     "and will be removed from pandas in a future version. "
   (...)
   9036     stacklevel=find_stack_level(),
   9037 )
-> 9039 return self._append(other, ignore_index, verify_integrity, sort)

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:9052, in DataFrame._append(self,
 other, ignore_index, verify_integrity, sort)
   9050 if isinstance(other, dict):
```

```
   9051     if not ignore_index:
-> 9052         raise TypeError("Can only append a dict if ignore_index=True")
   9053     other = Series(other)
   9054 if other.name is None and not ignore_index:

TypeError: Can only append a dict if ignore_index=True
```

Why are we getting an error here?

Its' saying the `ignore_index()` parameter needs to be set to True

In [62]:
```
new_row = {'Country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_
df = df.append(new_row, ignore_index=True)
df
```

```
C:\Users\kumar\AppData\Local\Temp\ipykernel_4240\1263752680.py:2: FutureWarning: The fra
me.append method is deprecated and will be removed from pandas in a future version. Use
pandas.concat instead.
  df = df.append(new_row, ignore_index=True)
```

Out[62]:

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

1705 rows × 5 columns

Perfect! So now our row is added at the bottom of the dataframe

But Please Note that:

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

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

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

What do you think this positional value would be?

In [63]:
```
df.loc[len(df.index)] = ['India',2000 ,13500000,"Asia",37.08,900.23]  # len(df.index) si
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
Input In [63], in <cell line: 1>()
----> 1 df.loc[len(df.index)] = ['India',2000 ,13500000,"Asia",37.08,900.23]

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:716, in _LocationIndexer.__setitem__(self, key, value)
    713 self._has_valid_setitem_indexer(key)
    715 iloc = self if self.name == "iloc" else self.obj.iloc
--> 716 iloc._setitem_with_indexer(indexer, value, self.name)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1682, in _iLocIndexer._setitem_with_indexer(self, indexer, value, name)
   1679     indexer, missing = convert_missing_indexer(indexer)
   1681     if missing:
-> 1682         self._setitem_with_indexer_missing(indexer, value)
   1683         return
   1685 # align and set the values

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1998, in _iLocIndexer._setitem_with_indexer_missing(self, indexer, value)
   1995     if is_list_like_indexer(value):
   1996         # must have conforming columns
   1997         if len(value) != len(self.obj.columns):
-> 1998             raise ValueError("cannot set a row with mismatched columns")
   2000     value = Series(value, index=self.obj.columns, name=indexer)
   2002 if not len(self.obj):
   2003     # We will ignore the existing dtypes instead of using
   2004     #  internals.concat logic

ValueError: cannot set a row with mismatched columns
```

In [64]:
```
df
```

Out[64]:

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

1705 rows × 5 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 [65]:
```
df.iloc[len(df.index)-1] = ['India', 2000,13500000,37.08,900.23]
df
```

Out[65]:

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

1705 rows × 5 columns

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

In [66]:
```
df.iloc[len(df.index)] = ['India', 2000,13500000,37.08,900.23]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [66], in <cell line: 1>()
----> 1 df.iloc[len(df.index)] = ['India', 2000,13500000,37.08,900.23]

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:713, in _LocationIndexer.__se
titem__(self, key, value)
    711     key = com.apply_if_callable(key, self.obj)
    712 indexer = self._get_setitem_indexer(key)
--> 713 self._has_valid_setitem_indexer(key)
    715 iloc = self if self.name == "iloc" else self.obj.iloc
    716 iloc._setitem_with_indexer(indexer, value, self.name)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1413, in _iLocIndexer._has_va
lid_setitem_indexer(self, indexer)
   1411 elif is_integer(i):
   1412     if i >= len(ax):
-> 1413         raise IndexError("iloc cannot enlarge its target object")
   1414 elif isinstance(i, dict):
   1415     raise IndexError("iloc cannot enlarge its target object")

IndexError: iloc cannot enlarge its target object
```

Why we are getting 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

Please Note:

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

Now what if we want to delete a row ?

Use df.drop()

If you remember we specified axis=1 for columns

We can modify this for rows

- We can use `axis=0` for rows

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

What do you think by looking at code for deleting column?

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

In [67]:
```
# Let's drop row with label 3
df = df.drop(3, axis=0)
df
```

Out[67]:

	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
4	Afghanistan	1972	13079460	36.088	739.981106
5	Afghanistan	1977	14880372	38.438	786.113360
...	...	...	...	...	...
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	India	2000	13500000	37.080	900.230000

1704 rows × 5 columns

Now we see that **row with label 3 is deleted**

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

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

In [68]:
```
df.loc[4] # The 4th row is printed
```

```
Country       Afghanistan
```

Out[68]:
```
year                 1972
population       13079460
life_exp            36.088
gdp_cap          739.981106
Name: 4, dtype: object
```

In [69]:
```
df.iloc[4] # The 5th row is printed
```

Out[69]:
```
Country        Afghanistan
year                  1977
population        14880372
life_exp             38.438
gdp_cap           786.11336
Name: 5, dtype: object
```

And hww can we drop multiple rows?

In [70]:
```
df.drop([1, 2, 4], axis=0) # drops rows with labels 1, 2, 4
```

Out[70]:

	Country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
5	Afghanistan	1977	14880372	38.438	786.113360
6	Afghanistan	1982	12881816	39.854	978.011439
7	Afghanistan	1987	13867957	40.822	852.395945
8	Afghanistan	1992	16317921	41.674	649.341395
...	...	...	...	...	...
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	India	2000	13500000	37.080	900.230000

1701 rows × 5 columns

Let's reset our indices now

In [71]:
```
df.reset_index(drop=True,inplace=True) # Since we removed a row earlier, we reset our in
```

Now 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 [72]:
```
df.loc[len(df.index)] = ['India',2000,13500000,37.08,900.23]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000,80.00,500.00]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000,80.00,500.00]
df.loc[len(df.index)] = ['India',2000 ,13500000,80.00,900.23]
df
```

Out[72]:

	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	1972	13079460	36.088	739.981106
4	Afghanistan	1977	14880372	38.438	786.113360
...	...	...	...	...	...
1703	India	2000	13500000	37.080	900.230000
1704	India	2000	13500000	37.080	900.230000
1705	Sri Lanka	2022	130000000	80.000	500.000000
1706	Sri Lanka	2022	130000000	80.000	500.000000
1707	India	2000	13500000	80.000	900.230000

1708 rows × 5 columns

Now how can we check for duplicate rows?

Use `duplicated()` method on the DataFrame

In [73]:
```
df.duplicated()
```

Out[73]:
```
0       False
1       False
2       False
3       False
4       False
        ...
1703    False
1704     True
1705    False
1706     True
1707    False
Length: 1708, dtype: bool
```

It outputs True if an entire row is identical to a previous row.

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

We can Pandas `loc` data selector to extract those duplicate rows

In [74]:
```
# Extract duplicate rows
df.loc[df.duplicated()]
```

Out[74]:

	Country	year	population	life_exp	gdp_cap
1704	India	2000	13500000	37.08	900.23
1706	Sri Lanka	2022	130000000	80.00	500.00

The first argument **df.duplicated()** will find the duplicate rows

The second argument `:` will display all columns

Now how can we remove these **duplicate rows** ?

We can use `drop_duplicates()` of Pandas for this

In [75]:
```
df.drop_duplicates()
```

Out[75]:

	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	1972	13079460	36.088	739.981106
4	Afghanistan	1977	14880372	38.438	786.113360
...	...	...	...	...	...
1701	Zimbabwe	2002	11926563	39.989	672.038623
1702	Zimbabwe	2007	12311143	43.487	469.709298
1703	India	2000	13500000	37.080	900.230000
1705	Sri Lanka	2022	130000000	80.000	500.000000
1707	India	2000	13500000	80.000	900.230000

1706 rows × 5 columns

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

Here we can use argument **keep**:

This Controls how to consider duplicate value.

It has only three distinct value

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

The default is 'first'.

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

In [76]:
```
df.drop_duplicates(keep='first')
```

Out[76]:

	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	1972	13079460	36.088	739.981106
4	Afghanistan	1977	14880372	38.438	786.113360
...	...	...	...	...	...
1701	Zimbabwe	2002	11926563	39.989	672.038623
1702	Zimbabwe	2007	12311143	43.487	469.709298

1703	India	2000	13500000	37.080	900.230000
1705	Sri Lanka	2022	130000000	80.000	500.000000
1707	India	2000	13500000	80.000	900.230000

1706 rows × 5 columns

If `last`, This considers last value as unique and rest of the same values as duplicate.

In [77]:
```
df.drop_duplicates(keep='last')
```

Out[77]:

	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	1972	13079460	36.088	739.981106
4	Afghanistan	1977	14880372	38.438	786.113360
...	...	...	...	...	...
1701	Zimbabwe	2002	11926563	39.989	672.038623
1702	Zimbabwe	2007	12311143	43.487	469.709298
1704	India	2000	13500000	37.080	900.230000
1706	Sri Lanka	2022	130000000	80.000	500.000000
1707	India	2000	13500000	80.000	900.230000

1706 rows × 5 columns

If `False`, this considers all of the same values as duplicates. All values are dropped.

In [78]:
```
df.drop_duplicates(keep=False)
```

Out[78]:

	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	1972	13079460	36.088	739.981106
4	Afghanistan	1977	14880372	38.438	786.113360
...	...	...	...	...	...
1699	Zimbabwe	1992	10704340	60.377	693.420786
1700	Zimbabwe	1997	11404948	46.809	792.449960
1701	Zimbabwe	2002	11926563	39.989	672.038623
1702	Zimbabwe	2007	12311143	43.487	469.709298
1707	India	2000	13500000	80.000	900.230000

1704 rows × 5 columns

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

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

In [79]:
```
df.drop_duplicates(subset=['Country'],keep='first')
```

Out[79]:

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

142 rows × 5 columns

In [80]:
```
df.drop_duplicates(subset=['Country', 'Continent'],keep='first')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [80], in <cell line: 1>()
----> 1 df.drop_duplicates(subset=['Country', 'Continent'],keep='first')

File ~\anaconda3\lib\site-packages\pandas\util\_decorators.py:311, in deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*args, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310     )
--> 311 return func(*args, **kwargs)

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:6116, in DataFrame.drop_duplicates(self, subset, keep, inplace, ignore_index)
   6114 inplace = validate_bool_kwarg(inplace, "inplace")
   6115 ignore_index = validate_bool_kwarg(ignore_index, "ignore_index")
-> 6116 duplicated = self.duplicated(subset, keep=keep)
   6118 result = self[-duplicated]
   6119 if ignore_index:

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:6250, in DataFrame.duplicated(self, subset, keep)
   6248 diff = Index(subset).difference(self.columns)
   6249 if not diff.empty:
-> 6250     raise KeyError(diff)
   6252 vals = (col.values for name, col in self.items() if name in subset)
   6253 labels, shape = map(list, zip(*map(f, vals)))
```

```
KeyError: Index(['Continent'], dtype='object')
```

Working with Rows and Columns together

In [81]:
```
import pandas as pd
import numpy as np
```

In [82]:
```
df = pd.read_csv('mckinsey.csv')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Input In [82], in <cell line: 1>()
----> 1 df = pd.read_csv('mckinsey.csv')

File ~\anaconda3\lib\site-packages\pandas\util\_decorators.py:311, in deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*args, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310     )
--> 311 return func(*args, **kwargs)

File ~\anaconda3\lib\site-packages\pandas\io\parsers\readers.py:680, in read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect, error_bad_lines, warn_bad_lines, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision, storage_options)
    665 kwds_defaults = _refine_defaults_read(
    666     dialect,
    667     delimiter,
   (...)
    676     defaults={"delimiter": ","},
    677 )
    678 kwds.update(kwds_defaults)
--> 680 return _read(filepath_or_buffer, kwds)

File ~\anaconda3\lib\site-packages\pandas\io\parsers\readers.py:575, in _read(filepath_or_buffer, kwds)
    572 _validate_names(kwds.get("names", None))
    574 # Create the parser.
--> 575 parser = TextFileReader(filepath_or_buffer, **kwds)
    577 if chunksize or iterator:
    578     return parser

File ~\anaconda3\lib\site-packages\pandas\io\parsers\readers.py:933, in TextFileReader.__init__(self, f, engine, **kwds)
    930     self.options["has_index_names"] = kwds["has_index_names"]
    932 self.handles: IOHandles | None = None
--> 933 self._engine = self._make_engine(f, self.engine)

File ~\anaconda3\lib\site-packages\pandas\io\parsers\readers.py:1217, in TextFileReader._make_engine(self, f, engine)
   1213     mode = "rb"
   1214 # error: No overload variant of "get_handle" matches argument types
   1215 # "Union[str, PathLike[str], ReadCsvBuffer[bytes], ReadCsvBuffer[str]]"
   1216 # , "str", "bool", "Any", "Any", "Any", "Any", "Any"
```

```
-> 1217 self.handles = get_handle(  # type: ignore[call-overload]
   1218     f,
   1219     mode,
   1220     encoding=self.options.get("encoding", None),
   1221     compression=self.options.get("compression", None),
   1222     memory map=self.options.get("memory_map", False),
   1223     is text=is text,
   1224     errors=self.options.get("encoding errors", "strict"),
   1225     storage_options=self.options.get("storage_options", None),
   1226 )
   1227 assert self.handles is not None
   1228 f = self.handles.handle

File ~\anaconda3\lib\site-packages\pandas\io\common.py:789, in get_handle(path_or_buf, m
ode, encoding, compression, memory_map, is_text, errors, storage_options)
    784 elif isinstance(handle, str):
    785     # Check whether the filename is to be opened in binary mode.
    786     # Binary mode does not support 'encoding' and 'newline'.
    787     if ioargs.encoding and "b" not in ioargs.mode:
    788         # Encoding
--> 789         handle = open(
    790             handle,
    791             ioargs.mode,
    792             encoding=ioargs.encoding,
    793             errors=errors,
    794             newline="",
    795         )
    796     else:
    797         # Binary mode
    798         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: 'mckinsey.csv'
```

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

We can use iloc

In [83]:
```
df.iloc[1:5, 1:4]
```

Out[83]:

	year	population	life_exp
1	1957	9240934	30.332
2	1962	10267083	31.997
3	1972	13079460	36.088
4	1977	14880372	38.438

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 [84]:
```
df.loc[1:5, 1:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [84], in <cell line: 1>()
----> 1 df.loc[1:5, 1:4]

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:961, in _LocationIndexer.__ge
```

```
titem__(self, key)
    959     if self._is_scalar_access(key):
    960         return self.obj._get_value(*key, takeable=self._takeable)
--> 961     return self._getitem_tuple(key)
    962 else:
    963     # we by definition only have the 0th axis
    964     axis = self.axis or 0

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1149, in _LocIndexer._getitem
_tuple(self, tup)
   1146 if self._multi_take_opportunity(tup):
   1147     return self._multi_take(tup)
-> 1149 return self._getitem_tuple_same_dim(tup)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:827, in _LocationIndexer._get
item_tuple_same_dim(self, tup)
    824 if com.is_null_slice(key):
    825     continue
--> 827 retval = getattr(retval, self.name)._getitem_axis(key, axis=i)
    828 # We should never have retval.ndim < self.ndim, as that should
    829 #  be handled by the _getitem_lowerdim call above.
    830 assert retval.ndim == self.ndim

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1180, in _LocIndexer._getitem
_axis(self, key, axis)
   1178 if isinstance(key, slice):
   1179     self._validate_key(key, axis)
-> 1180     return self._get_slice_axis(key, axis=axis)
   1181 elif com.is_bool_indexer(key):
   1182     return self._getbool_axis(key, axis=axis)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1214, in _LocIndexer._get_sli
ce_axis(self, slice_obj, axis)
   1211     return obj.copy(deep=False)
   1213 labels = obj._get_axis(axis)
-> 1214 indexer = labels.slice_indexer(slice_obj.start, slice_obj.stop, slice_obj.step)
   1216 if isinstance(indexer, slice):
   1217     return self.obj._slice(indexer, axis=axis)

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:6274, in Index.slice_inde
xer(self, start, end, step, kind)
   6231 """
   6232 Compute the slice indexer for input labels and step.
   6233
   (...)
   6270 slice(1, 3, None)
   6271 """
   6272 self._deprecated_arg(kind, "kind", "slice indexer")
-> 6274 start_slice, end_slice = self.slice_locs(start, end, step=step)
   6276 # return a slice
   6277 if not is_scalar(start_slice):

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:6484, in Index.slice_locs
(self, start, end, step, kind)
   6482 start_slice = None
   6483 if start is not None:
-> 6484     start_slice = self.get_slice_bound(start, "left")
   6485 if start_slice is None:
   6486     start_slice = 0

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:6393, in Index.get_slice_
bound(self, label, side, kind)
   6389 original_label = label
   6391 # For datetime indices label may be a string that has to be converted
   6392 # to datetime boundary according to its resolution.
-> 6393 label = self._maybe_cast_slice_bound(label, side)
```

```
   6395 # we need to look up the label
   6396 try:

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:6340, in Index._maybe_cast_slice_bound(self, label, side, kind)
   6335 # We are a plain index here (sub-class override this method if they
   6336 # wish to have special treatment for floats/ints, e.g. Float64Index and
   6337 # datetimelike Indexes
   6338 # reject them, if index does not contain label
   6339 if (is_float(label) or is_integer(label)) and label not in self:
-> 6340     raise self._invalid_indexer("slice", label)
   6342 return label

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

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

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

In [85]:
```
df.loc[1:5, ['country','life_exp']]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [85], in <cell line: 1>()
----> 1 df.loc[1:5, ['country','life_exp']]

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:961, in _LocationIndexer.__getitem__(self, key)
    959     if self._is_scalar_access(key):
    960         return self.obj._get_value(*key, takeable=self._takeable)
--> 961     return self._getitem_tuple(key)
    962 else:
    963     # we by definition only have the 0th axis
    964     axis = self.axis or 0

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1149, in _LocIndexer._getitem_tuple(self, tup)
   1146 if self._multi_take_opportunity(tup):
   1147     return self._multi_take(tup)
-> 1149 return self._getitem_tuple_same_dim(tup)

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:827, in _LocationIndexer._getitem_tuple_same_dim(self, tup)
    824 if com.is_null_slice(key):
    825     continue
--> 827 retval = getattr(retval, self.name)._getitem_axis(key, axis=i)
    828 # We should never have retval.ndim < self.ndim, as that should
    829 #  be handled by the _getitem_lowerdim call above.
    830 assert retval.ndim == self.ndim

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1191, in _LocIndexer._getitem_axis(self, key, axis)
   1188     if hasattr(key, "ndim") and key.ndim > 1:
   1189         raise ValueError("Cannot index with multidimensional key")
-> 1191     return self._getitem_iterable(key, axis=axis)
   1193 # nested tuple slicing
   1194 if is_nested_tuple(key, labels):

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1132, in _LocIndexer._getitem_iterable(self, key, axis)
   1129 self._validate_key(key, axis)
   1131 # A collection of keys
-> 1132 keyarr, indexer = self._get_listlike_indexer(key, axis)
   1133 return self.obj._reindex_with_indexers(
```

```
   1134     {axis: [keyarr, indexer]}, copy=True, allow_dups=True
   1135 )

File ~\anaconda3\lib\site-packages\pandas\core\indexing.py:1327, in _LocIndexer._get_listlike_indexer(self, key, axis)
   1324 ax = self.obj._get_axis(axis)
   1325 axis_name = self.obj._get_axis_name(axis)
-> 1327 keyarr, indexer = ax._get_indexer_strict(key, axis_name)
   1329 return keyarr, indexer

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:5782, in Index._get_indexer_strict(self, key, axis_name)
   5779 else:
   5780     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 5782 self._raise_if_missing(keyarr, indexer, axis_name)
   5784 keyarr = self.take(indexer)
   5785 if isinstance(key, Index):
   5786     # GH 42790 - Preserve name from an Index

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:5845, in Index._raise_if_missing(self, key, indexer, axis_name)
   5842     raise KeyError(f"None of [{key}] are in the [{axis_name}]")
   5844 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
-> 5845 raise KeyError(f"{not_found} not in index")

KeyError: "['country'] not in index"
```

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

In [86]:
```
df.loc[1:5, 'year':'population']
```

Out[86]:

	year	population
1	1957	9240934
2	1962	10267083
3	1972	13079460
4	1977	14880372
5	1982	12881816

How can we get specific rows and columns?

In [87]:
```
df.iloc[[0,10,100], [0,2,3]]
```

Out[87]:

	Country	population	life_exp
0	Afghanistan	8425333	28.801
10	Afghanistan	31889923	43.828
100	Bangladesh	80428306	46.923

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

Can we do step slicing?

Yes, just like we did in Numpy

In [88]:
```
df.iloc[1:10:2]
```

Out[88]:

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

Does step slicing work for loc too?

YES

In [89]:
```
df.loc[1:10:2]
```

Out[89]:

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

Pandas built-in operation

Let's select the feature `'life_exp'`

In [90]:
```
le = df['life_exp']
le
```

Out[90]:
```
0       28.801
1       30.332
2       31.997
3       36.088
4       38.438
         ...
1703    37.080
1704    37.080
1705    80.000
1706    80.000
1707    80.000
Name: life_exp, Length: 1708, dtype: float64
```

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

In [91]:
```
le.mean()
```

Out[91]:
```
59.499171358313774
```

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 [92]:
```
le.sum()
```

Out[92]:
```
101624.58468
```

In [93]:
```
le.count()
```

Out[93]:
```
1708
```

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

In [94]:
```
le.sum() / le.count()
```

Out[94]:
```
59.499171358313816
```

It gives the **mean** of life expectancy

Sorting

If you notice, `life_exp` col is not sorted

How can we perform sorting in pandas ?

In [95]:
```
df.sort_values(['life_exp'])
```

Out[95]:

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

1708 rows × 5 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 [96]:
```
df.sort_values(['life_exp'], ascending=False)
```

Out[96]:

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

1708 rows × 5 columns

Now the rows are sorted in **descending**

Can we do sorting on multiple columns?

YES

In [97]:
```
df.sort_values(['year', 'life_exp'])
```

Out[97]:

	Country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
551	Gambia	1952	284320	30.000	485.230659
35	Angola	1952	4232095	30.015	3520.610273
1343	Sierra Leone	1952	2143249	30.331	879.787736
1031	Mozambique	1952	6446316	31.286	468.526038
...	...	...	...	...	...
694	Iceland	2007	301931	81.757	36180.789190
670	Hong Kong, China	2007	6980412	82.208	39724.978670
802	Japan	2007	127467972	82.603	31656.068060
1705	Sri Lanka	2022	130000000	80.000	500.000000
1706	Sri Lanka	2022	130000000	80.000	500.000000

1708 rows × 5 columns

What exactly happened here?

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

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 [98]:
```
df.sort_values(['year', 'life_exp'], ascending=[False, True])
```

Out[98]:

	Country	year	population	life_exp	gdp_cap
1705	Sri Lanka	2022	130000000	80.000	500.000000
1706	Sri Lanka	2022	130000000	80.000	500.000000
1462	Swaziland	2007	1133066	39.613	4513.480643
1042	Mozambique	2007	19951656	42.082	823.685621
1690	Zambia	2007	11746035	42.384	1271.211593
...	...	...	...	...	...
407	Denmark	1952	4334000	70.780	9692.385245
1463	Sweden	1952	7124673	71.860	8527.844662
1079	Netherlands	1952	10381988	72.130	8941.571858
683	Iceland	1952	147962	72.490	7267.688428
1139	Norway	1952	3327728	72.670	10095.421720

1708 rows × 5 columns

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

Concatenating DataFrames

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

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

In [99]:
```
users = pd.DataFrame({"userid":[1, 2, 3], "name":["sharadh", "shahid", "khusalli"]})
users
```

Out[99]:

	userid	name
0	1	sharadh
1	2	shahid
2	3	khusalli

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

In [100…
```
msgs = pd.DataFrame({"userid":[1, 1, 2, 4], "msg":['hmm', "acha", "theek hai", "nice"]})
msgs
```

Out[100]:

	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 [101…
```
pd.concat([users, msgs])
```

Out[101]:

	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

Now how can we make the indices unique for each row?

In [102…
```
pd.concat([users, msgs], ignore_index = True)
```

Out[102]:

	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 [103…
```
pd.concat([users, msgs], axis=1)
```

Out[103]:

	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:

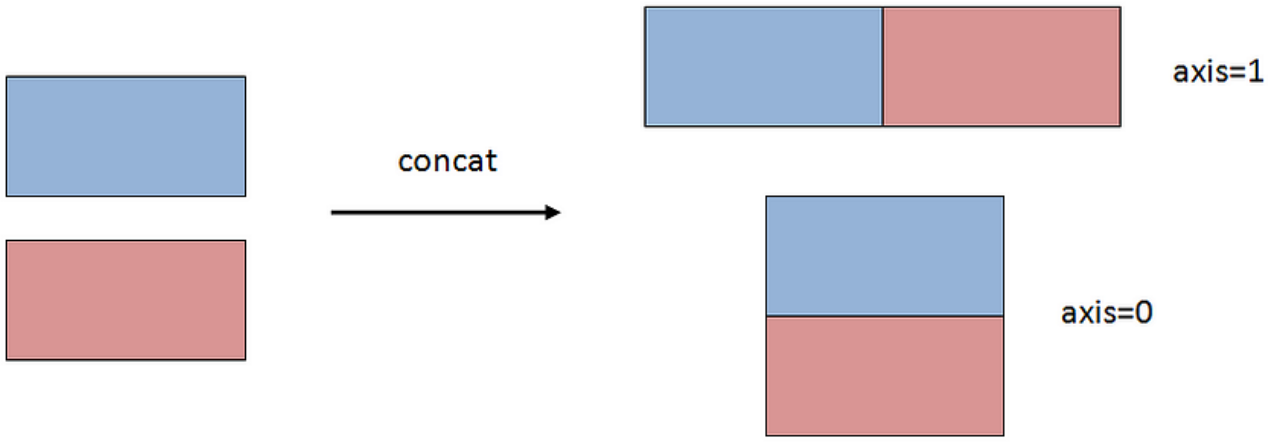
- Both the dataframes are combined horizontally (column-wise)
- It gives 2 columns with **different positional (implicit) index**, but **same label**

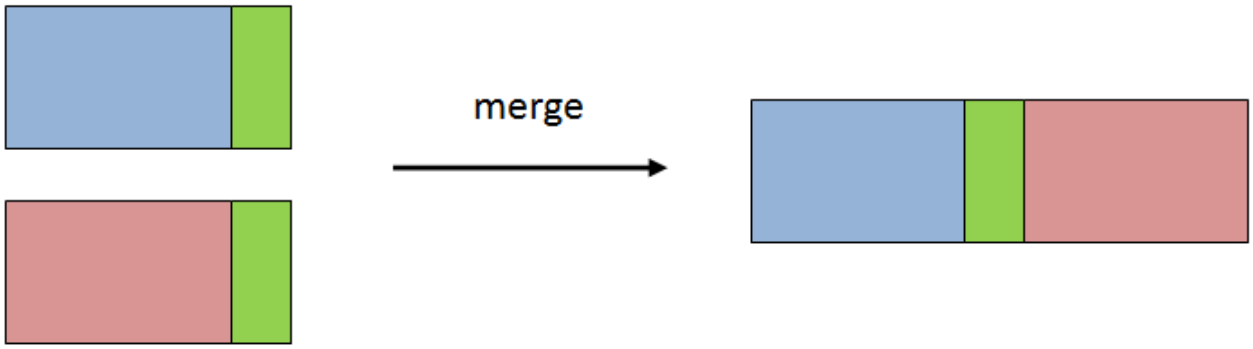
Merging Dataframes

So far we have only concatenated and not merged data

But what is the difference between concat and merge ?

- `concat`
  - simply stacks multiple DataFrame together along an axis
- `merge`
  - combines dataframes in a **smart** way based on values in shared columns





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 [104…
```
pd.concat([users, msgs], axis=1)
```

Out[104]:

	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 concat 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**

=> pd.concat() does not work according to the values in the columns

We need to **merge** the data

How can we join the dataframes ?

In [105…
```
users.merge(msgs, on="userid")
```

Out[105]:

	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

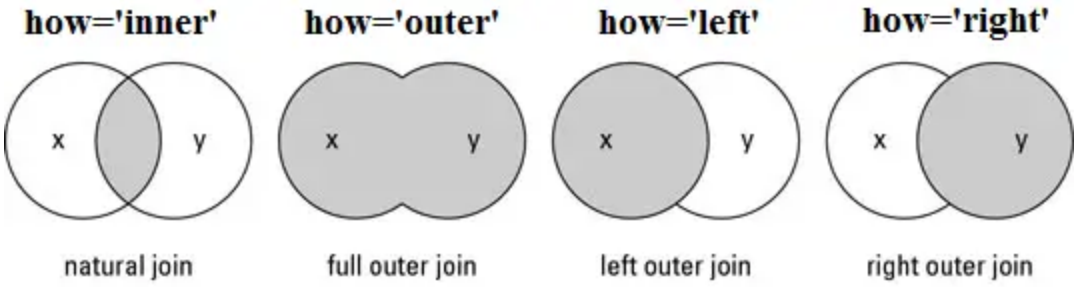
- When we **merge** these dataframes the **userid = 3 is not included**

- 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

Now what join we want to use to get info of all the users and all the messages?

In [106…
```
users.merge(msgs, on = "userid", how="outer")
```

Out[106]:

	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`

And what if we want the info of all the users in the dataframe?

In [107…
```
users.merge(msgs, on = "userid",how="left")
```

Out[107]:

	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 [108…
```
users.merge(msgs, on = "userid", how="right")
```

Out[108]:

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

Note,

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 [109…

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

Out[109]:

	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 name ?

In [110…

```
users.merge(msgs, left_on="id", right_on="userid")
```

Out[110]:

	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 here)
- `right_on` : Specifies the **key of the 2nd dataframe** (msgs here)

IMDB Movie Business Use-case

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 online database of information related to movies

The database contains info of several years about:

- Movies
- Rating
- Director
- Popularity
- Revenue & Budget

Lets download and read the IMDB dataset

- File1: https://drive.google.com/file/d/1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd/view?usp=sharing
- File2: https://drive.google.com/file/d/1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm/view?usp=sharing

In [111…
```
import pandas as pd
import numpy as np
```

In [112…
```
!gdown 1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
```

```
Downloading...
From: https://drive.google.com/uc?id=1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\movies.csv

  0%|          | 0.00/112k [00:00<?, ?B/s]
100%|##########| 112k/112k [00:00<00:00, 2.40MB/s]
```

In [113…
```
!gdown 1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
```

```
Downloading...
From: https://drive.google.com/uc?id=1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\directors.csv

  0%|          | 0.00/65.4k [00:00<?, ?B/s]
100%|##########| 65.4k/65.4k [00:00<00:00, 349kB/s]
100%|##########| 65.4k/65.4k [00:00<00:00, 349kB/s]
```

Here we have two csv files

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

In [114…
```
movies = pd.read_csv('movies.csv')
#Top 5 rows
movies.head()
```

Out[114]:

	Unnamed: 0	id	budget	popularity	revenue	title	vote_average	vote_count	director_id	year
0	0	43597	237000000	150	2787965087	Avatar	7.2	11800	4762	2009
1	1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	4763	2007
2	2	43599	245000000	107	880674609	Spectre	6.3	4466	4764	2015
3	3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	4765	2012
4	5	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	4767	2007

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

- **Top 10 most popular movies**, using `popularity`
- Or find some **highest rated movies**, using `vote_average`
- We can find number of **movies released per year** too
- Or maybe we can 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, 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 [115…
```
movies = pd.read_csv('movies.csv', index_col=0)
movies.head()
```

Out[115]:

	id	budget	popularity	revenue	title	vote_average	vote_count	director_id	year	month	
0	43597	237000000	150	2787965087	Avatar	7.2	11800	4762	2009	Dec	Th
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	4763	2007	May	Sa
2	43599	245000000	107	880674609	Spectre	6.3	4466	4764	2015	Oct	M
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	4765	2012	Jul	M
5	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	4767	2007	May	T

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

The default value is `index_col=None`

In [116…
```
movies.shape
```

Out[116]:
```
(1465, 11)
```

The movies df contains 1465 rows,11 columns

Lets read the `directors` dataset:

In [117…
```
directors = pd.read_csv('directors.csv',index_col=0)
directors.head()
```

Out[117]:

	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 [118…
```
directors.shape
```

Out[118]:
```
(2349, 3)
```

Directors df contains:

2349 rows,3 columns

Summary

1. Movie dataset contains info about **movies, release, popularity, ratings and the director ID**
2. Director dataset contains **detailed info about the director**

Merging the director and movie data

Now, how can we know the details about the Director of a particular movie?

We will have to merge these datasets

So on which column we should merge the dfs ?

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

If you observe,

=> `director_id` of movies are taken from `id` of directors dataframe

Thus we can merge our dataframes based on these two columns as **keys**

Before that, lets first check number of unique director values in our `movies` data

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

In [119…
```
movies['director_id'].nunique()
```

Out[119]:
```
199
```

Recall,

we had learnt about nunique earlier

Similarly for unique diretors in `directors` df

In [120…
```
directors['id'].nunique()
```

Out[120]:
```
2349
```

Summary:

- Movies Dataset: 1465 rows, but only 199 unique directors
- Directors Dataset: 2349 unique directors (= no of rows)

What can we infer from this?

=> Directors in `movies` is a subset of directors in `directors`

Now, how can we check if all `director_id` values are present in `id`?

In [121…
```
movies['director_id'].isin(directors['id'])
```

Out[121]:
```
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 the Dataframe column contains the specified value(s).

How is `isin` different from Python `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 df similar to the original df
- For rows with values of `director_id` present in `id` it returns True, else False

How can we check if there is any False here?

In [122…
```
np.all(movies['director_id'].isin(directors['id']))
```

Out[122]:
```
True
```

Lets finally merge our 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 ?

We can use LEFT JOIN

In [123…
```
data = movies.merge(directors, how='left', left_on='director_id',right_on='id')
data
```

Out[123]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	director_id	year	month
0	43597	237000000	150	2787965087	Avatar	7.2	11800	4762	2009	Dec
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	4763	2007	May
2	43599	245000000	107	880674609	Spectre	6.3	4466	4764	2015	Oct
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	4765	2012	Jul
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	4767	2007	May
...	...	...	...	...	...	...	...	...	...	...
1460	48363	0	3	321952	The Last Waltz	7.9	64	4809	1978	May
1461	48370	27000	19	3151130	Clerks	7.4	755	5369	1994	Sep
1462	48375	0	7	0	Rampage	6.0	131	5148	2009	Aug
1463	48376	0	3	0	Slacker	6.4	77	5535	1990	Jul
1464	48395	220000	14	2040920	El Mariachi	6.6	238	5097	1992	Sep

1465 rows × 14 columns

Notice, two stranger id columns `id_x` and `id_y`.

What do you think these newly created cols are?

Since the columns with name `id` is present in both the df

- `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 same as `director_id`
- But we dont require `director_id` further

So we can simply drop these features

In [124…
```
data.drop(['director_id','id_y'],axis=1,inplace=True)
data.head()
```

Out[124]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	dire

0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gor
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	Sa
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	C
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday	

Feature Exploration

Lets explore all the features in the merged dataset

In [125…
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1465 entries, 0 to 1464
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   id_x            1465 non-null   int64  
 1   budget          1465 non-null   int64  
 2   popularity      1465 non-null   int64  
 3   revenue         1465 non-null   int64  
 4   title           1465 non-null   object 
 5   vote_average    1465 non-null   float64
 6   vote_count      1465 non-null   int64  
 7   year            1465 non-null   int64  
 8   month           1465 non-null   object 
 9   day             1465 non-null   object 
 10  director_name   1465 non-null   object 
 11  gender          1341 non-null   object 
dtypes: float64(1), int64(6), object(5)
memory usage: 148.8+ KB
```

Looks like only `gender` column has missing values (will come later)

How can we describe these features to know more about their range of values?

In [126…
```
data.describe()
```

Out[126]:

	id_x	budget	popularity	revenue	vote_average	vote_count	year
count	1465.000000	1.465000e+03	1465.000000	1.465000e+03	1465.000000	1465.000000	1465.000000
mean	45225.191126	4.802295e+07	30.855973	1.432539e+08	6.368191	1146.396587	2002.615017
std	1189.096396	4.935541e+07	34.845214	2.064918e+08	0.818033	1578.077438	8.680141
min	43597.000000	0.000000e+00	0.000000	0.000000e+00	3.000000	1.000000	1976.000000
25%	44236.000000	1.400000e+07	11.000000	1.738013e+07	5.900000	216.000000	1998.000000
50%	45022.000000	3.300000e+07	23.000000	7.578164e+07	6.400000	571.000000	2004.000000

75%	45990.000000	6.600000e+07	41.000000	1.792469e+08	6.900000	1387.000000	2009.000000
max	48395.000000	3.800000e+08	724.000000	2.787965e+09	8.300000	13752.000000	2016.000000

This gives us all **statistical properties** of the columns

If you notice, some columns such as "title", "month" are missing

How are these **missing columns different?**

They are of **object dtype**

Then how can we include object type in `df.describe()`?

In [127…
```
data.describe(include=object)
```

Out[127]:

	title	month	day	director_name	gender
count	1465	1465	1465	1465	1341
unique	1465	12	7	199	2
top	Avatar	Dec	Friday	Steven Spielberg	Male
freq	1	193	654	26	1309

If you notice,

- The range of values in the `revenue` and `budget` seem to be very high
- Generally budget and revenue for Hollywood movies is in million dollars

How can we change the values of `revenue` and `budget` into million dollars USD?

In [128…
```
data['revenue'] = (data['revenue']/1000000).round(2)
data
```

Out[128]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	dire
0	43597	237000000	150	2787.97	Avatar	7.2	11800	2009	Dec	Thursday	
1	43598	300000000	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gor
2	43599	245000000	107	880.67	Spectre	6.3	4466	2015	Oct	Monday	Sa
3	43600	250000000	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	
4	43602	258000000	115	890.87	Spider-Man 3	5.9	3576	2007	May	Tuesday	
...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0	3	0.32	The Last Waltz	7.9	64	1978	May	Monday	
1461	48370	27000	19	3.15	Clerks	7.4	755	1994	Sep	Tuesday	

1462	48375	0	7	0.00	Rampage	6.0	131	2009	Aug	Friday
1463	48376	0	3	0.00	Slacker	6.4	77	1990	Jul	Friday
1464	48395	220000	14	2.04	El Mariachi	6.6	238	1992	Sep	Friday

1465 rows × 12 columns

Similarly, we can do it for 'budget' as well

In [129…
```
data['budget']=(data['budget']/1000000).round(2)
data.head()
```

Out[129]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_na
0	43597	237.0	150	2787.97	Avatar	7.2	11800	2009	Dec	Thursday	Jar Came
1	43598	300.0	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Verbi
2	43599	245.0	107	880.67	Spectre	6.3	4466	2015	Oct	Monday	Sam Men
3	43600	250.0	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Christop No
4	43602	258.0	115	890.87	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sam Ra

Fetching queries from dataframe

Lets say we are interested in fetching all **highly rated movies**

- say movies with **ratings > 7**

How can we get movies with ratings > 7?

We can use the concept of `masking`

Lets first create a mask to filter such movies

- In SQL: `SELECT * FROM movies WHERE vote_average > 7`
- In pandas:

In [130…
```
data['vote_average'] > 7
```

Out[130]:
```
0        True
1       False
2       False
3        True
4       False
        ...
1460     True
1461     True
1462    False
```

```
1463    False
1464    False
Name: vote_average, Length: 1465, dtype: bool
```

But we still don't know the row values ... Only that which row satisfied the condtion

How do we get the row values from this mask?

In [131…
```
data.loc[data['vote_average'] > 7]
```

Out[131]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	direc
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec	Thursday	
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	C
14	43616	250.00	120	956.02	The Hobbit: The Battle of the Five Armies	7.1	4760	2014	Dec	Wednesday	Pet
16	43619	250.00	94	958.40	The Hobbit: The Desolation of Smaug	7.6	4524	2013	Dec	Wednesday	Pet
19	43622	200.00	100	1845.03	Titanic	7.5	7562	1997	Nov	Tuesday	
...	...	...	...	...	...	...	...	...	...	...	
1456	48321	0.01	20	7.00	Eraserhead	7.5	485	1977	Mar	Saturday	D
1457	48323	0.00	5	0.00	The Mighty	7.1	51	1998	Oct	Friday	Pete
1458	48335	0.06	27	3.22	Pi	7.1	586	1998	Jul	Friday	
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May	Monday	
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep	Tuesday	K

301 rows × 12 columns

You can also perform the filtering without even using `loc`

In [132…
```
data[data['vote_average'] > 7]
```

Out[132]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	direc
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec	Thursday	
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	C
14	43616	250.00	120	956.02	The Hobbit:	7.1	4760	2014	Dec	Wednesday	Pet

					The Battle of the Five Armies						
16	43619	250.00	94	958.40	The Hobbit: The Desolation of Smaug	7.6	4524	2013	Dec	Wednesday	Pet
19	43622	200.00	100	1845.03	Titanic	7.5	7562	1997	Nov	Tuesday	
...	...	...	...	...	...	...	...	...	...	...	
1456	48321	0.01	20	7.00	Eraserhead	7.5	485	1977	Mar	Saturday	D
1457	48323	0.00	5	0.00	The Mighty	7.1	51	1998	Oct	Friday	Pete
1458	48335	0.06	27	3.22	Pi	7.1	586	1998	Jul	Friday	
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May	Monday	
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep	Tuesday	K

301 rows × 12 columns

But this is not recommended. Why ?

- It can create a confusion between implicit/explicit indexing used as discussed before
- `loc` is also much faster

Now, how can we return a subset of columns, say, only `title` and `director_name`?

In [133…
```
data.loc[data['vote_average'] > 7, ['title','director_name']]
```

Out[133]:

	title	director_name
0	Avatar	James Cameron
3	The Dark Knight Rises	Christopher Nolan
14	The Hobbit: The Battle of the Five Armies	Peter Jackson
16	The Hobbit: The Desolation of Smaug	Peter Jackson
19	Titanic	James Cameron
...	...	...
1456	Eraserhead	David Lynch
1457	The Mighty	Peter Chelsom
1458	Pi	Darren Aronofsky
1460	The Last Waltz	Martin Scorsese
1461	Clerks	Kevin Smith

301 rows × 2 columns

So far we saw only single condition for filtering

What if we want to filter highly rated movies released after 2014?

Notice that two conditions are involved here

1. Movies need to be highly rated i.e.. > 7
2. They should be 2015 and onwards

We can **use AND operator b/w multiple conditions**

In [134…
```
data.loc[(data['vote_average'] > 7) & (data['year'] >= 2015)].head()
```

Out[134]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	directo
30	43641	190.0	102	1506.25	Furious 7	7.3	4176	2015	Apr	Wednesday	Jam
78	43724	150.0	434	378.86	Mad Max: Fury Road	7.2	9427	2015	May	Wednesday	Georg
106	43773	135.0	100	532.95	The Revenant	7.3	6396	2015	Dec	Friday	Al G
162	43867	108.0	167	630.16	The Martian	7.6	7268	2015	Sep	Wednesday	Ridl
312	44128	75.0	48	108.15	The Man from U.N.C.L.E.	7.1	2265	2015	Aug	Thursday	Guy

Recall how we apply **mutliple conditions in numpy ?**

Use **elementwise operator `&` or `|`**

Note:

- **we cannot use `and` or `or`** with dataframe
- **for multiple conditions**, we need to put each **separate condition within parenthesis `()`**

Similarly how can we find movies released on either Friday or Sunday?

In [135…
```
data.loc[(data['day'] == 'Friday') | (data['day'] == 'Saturday')].head()
```

Out[135]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	directo
1	43598	300.0	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Ve
12	43614	380.0	135	1045.71	Pirates of the Caribbean: On Stranger Tides	6.4	4948	2011	May	Saturday	Rob M
22	43627	200.0	35	783.77	Spider-Man 2	6.7	4321	2004	Jun	Friday	Sar
25	43632	150.0	21	836.30	Transformers:	6.0	3138	2009	Jun	Friday	Mich

					Revenge of the Fallen						
40	43656	200.0	45	769.65	2012	5.6	4903	2009	Oct	Saturday	Em

Thus we can do complex queries using both `&` and `|` operators

Now let's try to answer few more Questions from this data

How will you find Top 5 most popular movies?

We can simply sort our data based on values of column 'popularity'

In [136…
```
data.sort_values(['popularity'],ascending=False).head(5)
```

Out[136]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	direct
58	43692	165.0	724	675.12	Interstellar	8.1	10867	2014	Nov	Wednesday	Ch
78	43724	150.0	434	378.86	Mad Max: Fury Road	7.2	9427	2015	May	Wednesday	Geo
119	43796	140.0	271	655.01	Pirates of the Caribbean: The Curse of the Bla...	7.5	6985	2003	Jul	Wednesday	Gore
120	43797	125.0	206	752.10	The Hunger Games: Mockingjay - Part 1	6.6	5584	2014	Nov	Tuesday	
45	43662	185.0	187	1004.56	The Dark Knight	8.2	12002	2008	Jul	Wednesday	Ch

On applying this to a string column, it sorts the dataframe ***lexicographically**

In [137…
```
data.sort_values(['title'],ascending=False).head(5)
```

Out[137]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	directo
436	44364	60.0	36	71.07	xXx: State of the Union	4.7	549	2005	Apr	Wednesday	Lee T
330	44165	70.0	46	277.45	xXx	5.8	1424	2002	Aug	Friday	Ro
994	45681	15.0	21	2.86	eXistenZ	6.7	475	1999	Apr	Wednesday	Cro
547	44594	50.0	37	55.97	Zoolander 2	4.7	797	2016	Feb	Saturday	B
850	45313	28.0	38	60.78	Zoolander	6.1	1337	2001	Sep	Friday	B

Now, how will get list of movies directed by a particular director, say, 'Christopher Nolan'?

In [138…
```
data.loc[data['director_name'] == 'Christopher Nolan',['title']]
```

Out[138]:

	title
3	The Dark Knight Rises
45	The Dark Knight
58	Interstellar
59	Inception
74	Batman Begins
565	Insomnia
641	The Prestige
1341	Memento

Note:

- The string indicating "Christopher Nolan" could have been something else as well.
- The better way is to use string methods, we will discuss this later

Apply

Now suppose we want to convert our `Gender` column data to numerical format

Basically,

- 0 for Male
- 1 for Female

How can we encode the column?

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

In [139…
```
def encode(data):
  if data == "Male":
    return 0
  else:
    return 1
```

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

In [140…
```
data['gender'] = data['gender'].apply(encode)
data
```

Out[140]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	directo
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec	Thursday	C
1	43598	300.00	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore V
2	43599	245.00	107	880.67	Spectre	6.3	4466	2015	Oct	Monday	Sam

3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Chri
4	43602	258.00	115	890.87	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sai
...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May	Monday	S
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep	Tuesday	Kevi
1462	48375	0.00	7	0.00	Rampage	6.0	131	2009	Aug	Friday	L
1463	48376	0.00	3	0.00	Slacker	6.4	77	1990	Jul	Friday	L
1464	48395	0.22	14	2.04	El Mariachi	6.6	238	1992	Sep	Friday	Ro

1465 rows × 12 columns

Notice how this is similar to using vectorization in Numpy

We thus can use `apply` to use a function throughout a column

Can we **use apply on multiple columns**?

Say,

How to find sum of revenue and budget per movie?

In [141…
```
data[['revenue', 'budget']].apply(np.sum)
```

Out[141]:
```
revenue    209867.04
budget      70353.62
dtype: float64
```

We can pass **multiple cols by packing them** within `[]`

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

But, we are getting the sum of the columns

How can we use apply to work on individual rows?

In [142…
```
data[['revenue', 'budget']].apply(np.sum, axis=1)
```

Out[142]:
```
0       3024.97
1       1261.00
2       1125.67
3       1334.94
4       1148.87
         ...
1460       0.32
1461       3.18
1462       0.00
1463       0.00
1464       2.26
Length: 1465, dtype: float64
```

Every row of `revenue` was added to same row of `budget`

What does this `axis` mean in apply ?

- If **axis = 0**, it will apply to **each column**, if **axis = 1**, **each row**
- By default axis = 0

=> `apply()` can be applied on any dataframe along any particular axis

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

In [143…
```
def prof(x): # We define a function to calculate profit
  return x['revenue']-x['budget']
data['profit'] = data[['revenue', 'budget']].apply(prof, axis = 1)
data
```

Out[143]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	directo
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec	Thursday	C
1	43598	300.00	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore V
2	43599	245.00	107	880.67	Spectre	6.3	4466	2015	Oct	Monday	Sam
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Chri
4	43602	258.00	115	890.87	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sa
...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May	Monday	S
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep	Tuesday	Kevi
1462	48375	0.00	7	0.00	Rampage	6.0	131	2009	Aug	Friday	L
1463	48376	0.00	3	0.00	Slacker	6.4	77	1990	Jul	Friday	L
1464	48395	0.22	14	2.04	El Mariachi	6.6	238	1992	Sep	Friday	Ro

1465 rows × 13 columns

Thus, we can access the columns by their names inside the functions too using apply

Importing Data

Let's first import our data and prepare it as we did in the last lecture

In [144…
```
import pandas as pd
import numpy as np
!gdown 1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
```

```
!gdown 1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
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='id')
data.drop(['director_id','id_y'],axis=1,inplace=True)
```

```
Downloading...
From: https://drive.google.com/uc?id=1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\movies.csv

  0%|          | 0.00/112k [00:00<?, ?B/s]
100%|##########| 112k/112k [00:00<00:00, 508kB/s]
100%|##########| 112k/112k [00:00<00:00, 508kB/s]
Downloading...
From: https://drive.google.com/uc?id=1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\directors.csv

  0%|          | 0.00/65.4k [00:00<?, ?B/s]
100%|##########| 65.4k/65.4k [00:00<00:00, 296kB/s]
100%|##########| 65.4k/65.4k [00:00<00:00, 296kB/s]
```

Grouping

How can we know the number of movies released by a particular director, say, Christopher Nolan?

In [145…
```
data.loc[data['director_name'] == 'Christopher Nolan',['title']].count()
```

Out[145]:
```
title    8
dtype: int64
```

What if we have to do find number of movies of each director?

We have value_counts() for this

In [146…
```
data["director_name"].value_counts()
```

Out[146]:
```
Steven Spielberg      26
Martin Scorsese       19
Clint Eastwood        19
Woody Allen           18
Ridley Scott          16
                      ..
Tim Hill               5
Jonathan Liebesman     5
Roman Polanski         5
Larry Charles          5
Nicole Holofcener      5
Name: director_name, Length: 199, dtype: int64
```

How does this exactly work?

We can assume pandas must have **grouped the rows internally** to find the count

But what if we need to find some **other metric** besides count?

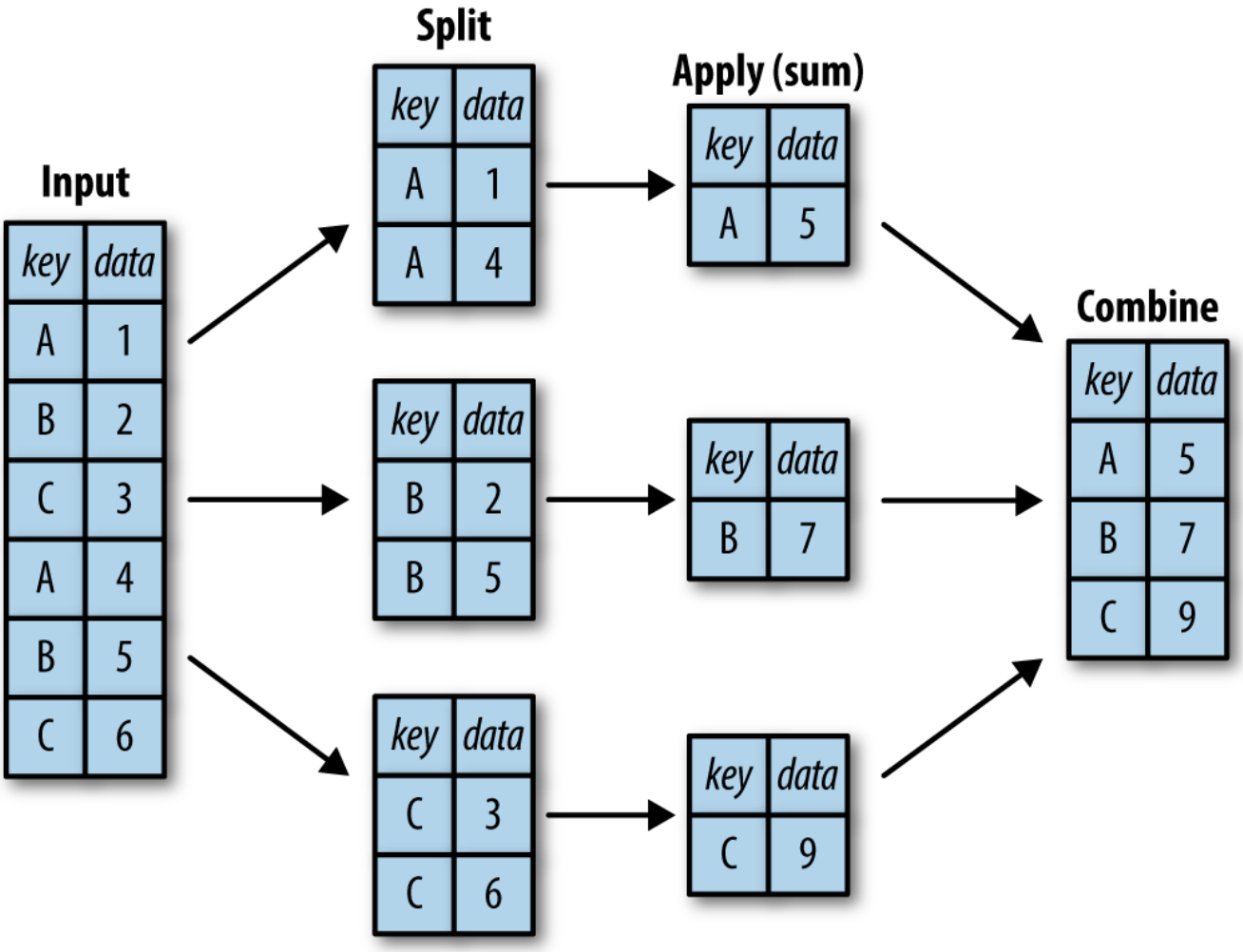
For example, **average popularity** of each director, or **max rating** among all movies by a director?

How can you find the average popularity of each director?

We will have to some group our rows director wise.

What is Grouping ?

Simply it could be understood through the terms - Split, apply, combine



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

Note:

All these steps are to understand the topic

Group based Aggregates

Now, how can we group our data director-wise?

In [147…
```
data.groupby('director_name')
```

Out[147]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001A2D0D84A30>
```

Notice,

- It's a **DataFrameGroupBy type object**

- **NOT a DataFrame** type object

What is `groupby('director_name')` doing?

Grouping all rows in which **director_name** value is **same**

But it's returning an object, we would want to get information out of this object.

Let's look at few attributes of the same.

How can we know the number of groups our data is divided into?

In [148…
```
data.groupby('director_name').ngroups
```

Out[148]:
```
199
```

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

In [149…
```
data.groupby('director_name').groups
```

Out[149]:
```
{'Adam McKay': [176, 323, 366, 505, 839, 916], 'Adam Shankman': [265, 300, 350, 404, 45
8, 843, 999, 1231], 'Alejandro González Iñárritu': [106, 749, 1015, 1034, 1077, 1405],
'Alex Proyas': [95, 159, 514, 671, 873], 'Alexander Payne': [793, 1006, 1101, 1211, 128
1], 'Andrew Adamson': [11, 43, 328, 501, 947], 'Andrew Niccol': [533, 603, 701, 722, 143
9], 'Andrzej Bartkowiak': [349, 549, 754, 911, 924], 'Andy Fickman': [517, 681, 909, 92
6, 973, 1023], 'Andy Tennant': [314, 320, 464, 593, 676, 885], 'Ang Lee': [99, 134, 748,
840, 1089, 1110, 1132, 1184], 'Anne Fletcher': [610, 650, 736, 789, 1206], 'Antoine Fuqu
a': [310, 338, 424, 467, 576, 808, 818, 1105], 'Atom Egoyan': [946, 1128, 1164, 1194, 13
47, 1416], 'Barry Levinson': [313, 319, 471, 594, 878, 898, 1013, 1037, 1082, 1143, 118
5, 1345, 1378], 'Barry Sonnenfeld': [13, 48, 90, 205, 591, 778, 783], 'Ben Stiller': [20
9, 212, 547, 562, 850], 'Bill Condon': [102, 307, 902, 1233, 1381], 'Bobby Farrelly': [3
52, 356, 481, 498, 624, 630, 654, 806, 928, 972, 1111], 'Brad Anderson': [1163, 1197, 13
50, 1419, 1430], 'Brett Ratner': [24, 39, 188, 207, 238, 292, 405, 456, 920], 'Brian De
Palma': [228, 255, 318, 439, 747, 905, 919, 1088, 1232, 1261, 1317, 1354], 'Brian Helgel
and': [512, 607, 623, 742, 933], 'Brian Levant': [418, 449, 568, 761, 860, 1003], 'Brian
Robbins': [416, 441, 669, 962, 988, 1115], 'Bryan Singer': [6, 32, 33, 44, 122, 216, 29
7, 1326], 'Cameron Crowe': [335, 434, 488, 503, 513, 698], 'Catherine Hardwicke': [602,
695, 724, 937, 1406, 1412], 'Chris Columbus': [117, 167, 204, 218, 229, 509, 656, 897, 9
96, 1086, 1129], 'Chris Weitz': [17, 500, 794, 869, 1202, 1267], 'Christopher Nolan':
[3, 45, 58, 59, 74, 565, 641, 1341], 'Chuck Russell': [177, 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], 'Da
nny Boyle': [527, 668, 1083, 1085, 1126, 1168, 1287, 1385], 'Darren Aronofsky': [113, 75
1, 1187, 1328, 1363, 1458], 'Darren Lynn Bousman': [1241, 1243, 1283, 1338, 1440], 'Davi
d Ayer': [50, 273, 741, 1024, 1146, 1407], 'David Cronenberg': [541, 767, 994, 1055, 125
4, 1268, 1334], 'David Fincher': [62, 213, 253, 383, 398, 478, 522, 555, 618, 785], 'Dav
id Gordon Green': [543, 862, 884, 927, 1376, 1418, 1432, 1459], 'David Koepp': [443, 64
4, 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],
'David Zucker': [569, 619, 965, 1052, 1175], 'Dennis Dugan': [217, 260, 267, 293, 303, 7
18, 780, 977, 1247], 'Donald Petrie': [427, 507, 570, 649, 858, 894, 1106, 1331], 'Doug
Liman': [52, 148, 251, 399, 544, 1318, 1451], 'Edward Zwick': [92, 182, 346, 566, 791, 8
19, 825], 'F. Gary Gray': [308, 402, 491, 523, 697, 833, 1272, 1380], 'Francis Ford Copp
ola': [487, 559, 622, 646, 772, 1076, 1155, 1253, 1312], 'Francis Lawrence': [63, 72, 10
9, 120, 679], 'Frank Coraci': [157, 249, 275, 451, 577, 599, 963], 'Frank Oz': [193, 35
5, 473, 580, 712, 813, 987], 'Garry Marshall': [329, 496, 528, 571, 784, 893, 1029, 116
9], 'Gary Fleder': [518, 667, 689, 867, 981, 1165], 'Gary Winick': [258, 797, 798, 804,
1454], 'Gavin O'Connor': [820, 841, 939, 953, 1444], 'George A. Romero': [250, 1066, 109
6, 1278, 1367, 1396], 'George Clooney': [343, 450, 831, 966, 1302], 'George Miller': [7
8, 103, 233, 287, 1250, 1403, 1450], 'Gore Verbinski': [1, 8, 9, 107, 119, 633, 1040],
'Guillermo del Toro': [35, 252, 419, 486, 1118], 'Gus Van Sant': [595, 1018, 1027, 1159,
1240, 1311, 1398], 'Guy Ritchie': [124, 215, 312, 1093, 1225, 1269, 1420], 'Harold Rami
```

```
s': [425, 431, 558, 586, 788, 1137, 1166, 1325], 'Ivan Reitman': [274, 643, 816, 883, 91
0, 935, 1134, 1242], 'James Cameron': [0, 19, 170, 173, 344, 1100, 1320], 'James Ivory':
[1125, 1152, 1180, 1291, 1293, 1390, 1397], 'James Mangold': [140, 141, 557, 560, 829, 8
45, 958, 1145], 'James Wan': [30, 617, 1002, 1047, 1337, 1417, 1424], 'Jan de Bont': [15
5, 224, 231, 270, 781], 'Jason Friedberg': [812, 1010, 1012, 1014, 1036], 'Jason Reitma
n': [792, 1092, 1213, 1295, 1299], 'Jaume Collet-Serra': [516, 540, 640, 725, 1011, 118
9], 'Jay Roach': [195, 359, 389, 397, 461, 703, 859, 1072], 'Jean-Pierre Jeunet': [423,
485, 605, 664, 765], 'Joe Dante': [284, 525, 638, 1226, 1298, 1428], 'Joe Wright': [85,
432, 553, 803, 814, 855], 'Joel Coen': [428, 670, 691, 707, 721, 889, 906, 980, 1157, 12
38, 1305], 'Joel Schumacher': [128, 184, 348, 484, 572, 614, 652, 764, 876, 886, 1108, 1
230, 1280], 'John Carpenter': [537, 663, 686, 861, 938, 1028, 1080, 1102, 1329, 1371],
'John Glen': [601, 642, 801, 847, 864], 'John Landis': [524, 868, 1276, 1384, 1435], 'Jo
hn Madden': [457, 882, 1020, 1249, 1257], 'John McTiernan': [127, 214, 244, 351, 534, 56
3, 648, 782, 838, 1074], 'John Singleton': [294, 489, 732, 796, 1120, 1173, 1316], 'John
Whitesell': [499, 632, 763, 1119, 1148], 'John Woo': [131, 142, 264, 371, 420, 675, 118
2], 'Jon Favreau': [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 Demme': [277, 493,
1000, 1123, 1215], 'Jonathan Liebesman': [81, 143, 339, 1117, 1301], 'Judd Apatow': [32
1, 710, 717, 865, 881], 'Justin Lin': [38, 123, 246, 1437, 1447], 'Kenneth Branagh': [8
0, 197, 421, 879, 1094, 1277, 1288], 'Kenny Ortega': [412, 852, 1228, 1315, 1365], 'Kevi
n Reynolds': [53, 502, 639, 1019, 1059], ...}
```

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

In [150…
```
data.groupby('director_name').get_group('Alexander Payne')
```

Out[150]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	d
793	45163	30000000	19	105834556	About Schmidt	6.7	362	2002	Dec	Friday	
1006	45699	20000000	40	177243185	The Descendants	6.7	934	2011	Sep	Friday	
1101	46004	16000000	23	109502303	Sideways	6.9	478	2004	Oct	Friday	
1211	46446	12000000	29	17654912	Nebraska	7.4	636	2013	Sep	Saturday	
1281	46813	0	13	0	Election	6.7	270	1999	Apr	Friday	

Great! We are able to extract the data from our DataFrameGroupBy object

But can we extend this to finding an aggregate metric of the data?

How can we find count of each director?

This does give us the max value of the data, but for **all the features**

In [151…
```
data.groupby('director_name')['title'].count()
```

Out[151]:
```
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
```

Now say we want to know two aggregations for any feature.

For e.g., the very first year and the latest year a director released a movie

This is basically the `min` and `max` of `year` column, grouped by director

How can we find multiple aggregations of any feature?

In [152…
```
data.groupby(['director_name'])["year"].aggregate(['min', 'max'])
```

Out[152]:

director_name	min	max
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

Group based Filtering

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

Lets assume,

- high budget director -> any director with **atleast one movie with budget >100M**

We can get the highest budget movie data of every director

In [153…
```
data_dir_budget = data.groupby("director_name")["budget"].max().reset_index()
data_dir_budget.head()
```

Out[153]:

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

4	Alexander Payne	30000000

How can we **filter** out the director names with **max budget >100M**?

In [154…
```
names = data_dir_budget.loc[data_dir_budget["budget"] >= 100, "director_name"]
```

Finally, how can we filter out the details of the movies by these directors?

In [155…
```
data.loc[data['director_name'].isin(names)]
```

Out[155]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	c
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday	
...	...	...	...	...	...	...	...	...	...	...	...
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday	
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday	
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday	
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday	
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday	

1465 rows × 12 columns

Recall `isin()` from last lecture

Can we do filtering of groups in a single go?

YES

In [156…
```
data.groupby('director_name').filter(lambda x: x["budget"].max() >= 100)
```

Out[156]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	c
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	
1	43598	300000000	139	961000000	Pirates of the Caribbean:	6.9	4500	2007	May	Saturday	

					At World's End					
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday
...	...	...	...	...	...	...	...	...	...	...
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday

1465 rows × 12 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 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

Now let's assume, we call a movi risky if,

- its budget is higher than the average revenue of its director

How do we filter risky movies?

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

Can we use `apply` here?

Yes!

How do we use apply for this column?

- We will define a function to compute the subtraction
- Pass this function in `apply`

In [157…
```
def func(x):
  x["risky"] = x["budget"] - x["revenue"].mean() >= 0
  return x
data_risky = data.groupby("director_name").apply(func)
data_risky
```

Out[157]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday
...	...	...	...	...	...	...	...	...	...	...
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday

1465 rows × 13 columns

Recall `apply()` from our earlier lectures

What did we do here?

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

Lets see if there are any risky movies

In [158…
```
data_risky.loc[data_risky["risky"]]
```

Out[158]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day
7	43608	200000000	107	586090727	Quantum of Solace	6.1	2965	2008	Oct	Thursday
12	43614	380000000	135	1045713802	Pirates of the	6.4	4948	2011	May	Saturday

					Caribbean: On Stranger Tides					
15	43618	200000000	37	310669540	Robin Hood	6.2	1398	2010	May	Wednesday
20	43624	209000000	64	303025485	Battleship	5.5	2114	2012	Apr	Wednesday
24	43630	210000000	3	459359555	X-Men: The Last Stand	6.3	3525	2006	May	Wednesday
...	...	...	...	...	...	...	...	...	...	...
1347	47224	5000000	7	3263585	The Sweet Hereafter	6.8	103	1997	May	Wednesday
1349	47229	5000000	3	4842699	90 Minutes in Heaven	5.4	40	2015	Sep	Friday
1351	47233	5000000	6	0	Light Sleeper	5.7	15	1992	Aug	Friday
1356	47263	15000000	10	0	Dying of the Light	4.5	118	2014	Dec	Thursday
1383	47453	3500000	4	0	In the Name of the King III	3.3	19	2013	Dec	Friday

131 rows × 13 columns

Yes, there are some 131 movies whose budget was **greater than average** earnings of its director

Multi-Indexing

Now, lets say, you want to find who is the **most productive director**

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

- Will you decide based on the **number of movies** released by a director?

Or

- will consider **quality into consideration also?**

Or

- will you also consider the amount of business the movie is doing?

To simplify,

Lets calculate who has directed maximum number of movies

In [159…
```
data.groupby(['director_name'])['title'].count().sort_values(ascending=False)
```

Out[159]:
```
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
```

Looks like `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 other directors

How would you calculate active years for **every director**?

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

How can we calculate multiple aggregates such as `min` and `max`, along with count of `titles` together?

In [160…
```
data_agg = data.groupby(['director_name'])[["year", "title"]].aggregate({"year":['min','
data_agg
```

Out[160]:

	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 **Multi-index Dataframe**

What is Multi-index Dataframe ?

- It can have **multiple indexes along a dimension**
  - no of dimensions remain same though => 2D
- Multi-level indexes are **possible both for rows and columns**

In [161…
```
data_agg.columns #Printing the columns for better clarity
```

Out[161]:
```
MultiIndex([( 'year',   'min'),
            ( 'year',   'max'),
            ('title', 'count')],
           )
```

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

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

In [162…
```
data_agg["year"]
```

Out[162]:

director_name	min	max
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?

Example: `year_min` , `year_max` , `title_count`

In [163…
```
data_agg.columns = ['_'.join(col) for col in data_agg.columns]
data_agg
```

Out[163]:

director_name	year_min	year_max	title_count
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 [164…
```
data.groupby('director_name')[['year', 'title']].aggregate(
    year_max=('year','max'),
    year_min=('year','min'),
    title_count=('title','count')
)
```

Out[164]:

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

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

How can we convert row labels into a column?

In [165…
```
data_agg.reset_index()
```

Out[165]:

	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

Recall,

We learnt `reset_index()` earlier

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

First calculate how many years the director has been active.

In [166…
```
data_agg["yrs_active"] = data_agg["year_max"] - data_agg["year_min"]
data_agg
```

Out[166]:

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

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

In [167…
```
data_agg["movie_per_yr"] = data_agg["title_count"] / data_agg["yrs_active"]
data_agg
```

Out[167]:

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

Now finally sort the values

In [168…
```
data_agg.sort_values("movie_per_yr", ascending=False)
```

Out[168]:

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 the **truly most productive director**

Importing our data

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

Link: https://drive.google.com/file/d/173A59xh2mnpmljCCB9bhC4C5eP2IS6qZ/view?usp=sharing

In [169…
```
!gdown 173A59xh2mnpmljCCB9bhC4C5eP2IS6qZ
```

```
Downloading...
From: https://drive.google.com/uc?id=173A59xh2mnpmljCCB9bhC4C5eP2IS6qZ
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\Pfizer_1.csv

  0%|          | 0.00/1.51k [00:00<?, ?B/s]
100%|##########| 1.51k/1.51k [00:00<00:00, 1.48MB/s]
```

What is the data about?

- Temperature (K)
- Pressure (P)

are recorded after an **interval of 1 hour** everyday to monitor the drug stability in a drug development test

==> These data points are thus used to **identify the optimal set of values of parameters** for the stability of the drugs

Now, Let's explore this dataset

In [170…
```
import pandas as pd
import numpy as np
```

In [171…
```
data = pd.read_csv('Pfizer_1.csv')
```

In [172…
```
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
```

In [173…
```
data.shape
```

Out[173]:
```
(18, 15)
```

In [174…
```
data.head()
```

Out[174]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30:0
0	15-10-	diltiazem hydrochloride	Temperature	23.0	22.0	NaN	21.0	21.0	22	23.0	21.0	22

	2020											
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	NaN	11.0	13.0	14	16.0	16.0	24
2	15-10-2020	docetaxel injection	Temperature	NaN	17.0	18.0	NaN	17.0	18	NaN	NaN	23
3	15-10-2020	docetaxel injection	Pressure	NaN	22.0	22.0	NaN	22.0	23	NaN	NaN	27
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	NaN	NaN	27.0	NaN	26	25.0	24.0	23

In [175…
```
data.tail()
```

Out[175]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30
13	17-10-2020	diltiazem hydrochloride	Pressure	3.0	4.0	4.0	4.0	6.0	8	9.0	NaN	
14	17-10-2020	docetaxel injection	Temperature	12.0	13.0	14.0	15.0	16.0	17	18.0	19.0	2
15	17-10-2020	docetaxel injection	Pressure	20.0	22.0	22.0	22.0	22.0	23	25.0	26.0	2
16	17-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	16.0	17.0	18	19.0	20.0	2
17	17-10-2020	ketamine hydrochloride	Pressure	8.0	9.0	10.0	11.0	11.0	12	12.0	11.0	1

Melting in Pandas

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" timestamp columns into two columns** - timestamp and corresponding values

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

In [176…
```
pd.melt(data, id_vars=['Date', 'Parameter', 'Drug_Name'])
```

Out[176]:

	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 the `id_vars are set of variables which remain unmelted

How does `pd.melt()` work?

- Pass in the **DataFrame**
- Pass in the **column names to not 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 [177…
```
data_melt = pd.melt(data,id_vars = ['Date', 'Drug_Name', 'Parameter'],
                    var_name = "time",
                    value_name = 'reading')

data_melt
```

Out[177]:

	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 values in column `reading`

- 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 **value** column

Pivot

Now suppose we want to convert our data back to **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, before it was melted?

In [178…
```
data_melt.pivot(index=['Date','Drug_Name','Parameter'],  # Column to use to make new fra
                columns = 'time',                       # Column to use to make new frame's c
                values='reading')                        # Columns to use for populating new
```

Out[178]:

		time	10:30:00	11:30:00	12:30:00	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:3
Date	**Drug_Name**	**Parameter**									
15-10-2020	**diltiazem hydrochloride**	**Pressure**	18.0	19.0	20.0	12.0	13.0	NaN	11.0	13.0	
		Temperature	20.0	20.0	21.0	23.0	22.0	NaN	21.0	21.0	
	docetaxel injection	**Pressure**	26.0	29.0	28.0	NaN	22.0	22.0	NaN	22.0	
		Temperature	23.0	25.0	25.0	NaN	17.0	18.0	NaN	17.0	
	ketamine hydrochloride	**Pressure**	9.0	9.0	11.0	8.0	NaN	NaN	7.0	NaN	
		Temperature	22.0	21.0	20.0	24.0	NaN	NaN	27.0	NaN	
16-10-2020	**diltiazem hydrochloride**	**Pressure**	24.0	NaN	27.0	18.0	19.0	20.0	21.0	22.0	
		Temperature	40.0	NaN	42.0	34.0	35.0	36.0	36.0	37.0	
	docetaxel	**Pressure**	28.0	29.0	30.0	23.0	24.0	NaN	25.0	26.0	

	injection	Temperature	56.0	57.0	58.0	46.0	47.0	NaN	48.0	48.0
	ketamine hydrochloride	Pressure	16.0	17.0	18.0	12.0	12.0	13.0	NaN	15.0
		Temperature	13.0	14.0	15.0	8.0	9.0	10.0	NaN	11.0
17-10-2020	diltiazem hydrochloride	Pressure	11.0	13.0	14.0	3.0	4.0	4.0	4.0	6.0
		Temperature	14.0	11.0	10.0	20.0	19.0	19.0	18.0	17.0
	docetaxel injection	Pressure	28.0	29.0	28.0	20.0	22.0	22.0	22.0	22.0
		Temperature	21.0	22.0	23.0	12.0	13.0	14.0	15.0	16.0
	ketamine hydrochloride	Pressure	13.0	14.0	15.0	8.0	9.0	10.0	11.0	11.0
		Temperature	22.0	23.0	24.0	13.0	14.0	15.0	16.0	17.0

Notice,

We are getting **multiple indices** here

How can we reset this to a single-index dataframe?

In [179…
```
data_melt.pivot(index=['Date','Drug_Name','Parameter'],
                columns = 'time',
                values='reading').reset_index()
```

Out[179]:

time	Date	Drug_Name	Parameter	10:30:00	11:30:00	12:30:00	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00
0	15-10-2020	diltiazem hydrochloride	Pressure	18.0	19.0	20.0	12.0	13.0	NaN	11.0	13.0
1	15-10-2020	diltiazem hydrochloride	Temperature	20.0	20.0	21.0	23.0	22.0	NaN	21.0	21.0
2	15-10-2020	docetaxel injection	Pressure	26.0	29.0	28.0	NaN	22.0	22.0	NaN	22.0
3	15-10-2020	docetaxel injection	Temperature	23.0	25.0	25.0	NaN	17.0	18.0	NaN	17.0
4	15-10-2020	ketamine hydrochloride	Pressure	9.0	9.0	11.0	8.0	NaN	NaN	7.0	NaN
5	15-10-2020	ketamine hydrochloride	Temperature	22.0	21.0	20.0	24.0	NaN	NaN	27.0	NaN
6	16-10-2020	diltiazem hydrochloride	Pressure	24.0	NaN	27.0	18.0	19.0	20.0	21.0	22.0
7	16-10-2020	diltiazem hydrochloride	Temperature	40.0	NaN	42.0	34.0	35.0	36.0	36.0	37.0
8	16-10-2020	docetaxel injection	Pressure	28.0	29.0	30.0	23.0	24.0	NaN	25.0	26.0
9	16-	docetaxel	Temperature	56.0	57.0	58.0	46.0	47.0	NaN	48.0	48.0

	10-2020	injection									
10	16-10-2020	ketamine hydrochloride	Pressure	16.0	17.0	18.0	12.0	12.0	13.0	NaN	15.0
11	16-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	8.0	9.0	10.0	NaN	11.0
12	17-10-2020	diltiazem hydrochloride	Pressure	11.0	13.0	14.0	3.0	4.0	4.0	4.0	6.0
13	17-10-2020	diltiazem hydrochloride	Temperature	14.0	11.0	10.0	20.0	19.0	19.0	18.0	17.0
14	17-10-2020	docetaxel injection	Pressure	28.0	29.0	28.0	20.0	22.0	22.0	22.0	22.0
15	17-10-2020	docetaxel injection	Temperature	21.0	22.0	23.0	12.0	13.0	14.0	15.0	16.0
16	17-10-2020	ketamine hydrochloride	Pressure	13.0	14.0	15.0	8.0	9.0	10.0	11.0	11.0
17	17-10-2020	ketamine hydrochloride	Temperature	22.0	23.0	24.0	13.0	14.0	15.0	16.0	17.0

==> `pivot()` is the exact opposite of melt

How does `pivot()` work?

- Column `Time` is pivoted upon `Date`, `Drug_Name` and `Parameter`

In [180…
```
data_melt.head()
```

Out[180]:

	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 really suitable

- We want to **split one single column into multiple columns**

How can we divide the Parameter column again?

In [181…
```
data_tidy = data_melt.pivot(index=['Date','time', 'Drug_Name'],
                                        columns = 'Parameter',
                                        values='reading')

data_tidy
```

Out[181]:

		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

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

In [182…
```
data_tidy = data_tidy.reset_index()
data_tidy
```

Out[182]:

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 [183…
```
data_tidy.columns.name = 'None'
```

In [184…
```
data_tidy.head()
```

Out[184]:

None	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

In [185…
```
pd.pivot_table?
```

In [ ]:
```

```

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 [186…
```
data_tidy.pivot(index=['Drug_Name'],
                columns = 'Date',
                values=['Temperature'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [186], in <cell line: 1>()
----> 1 data_tidy.pivot(index=['Drug_Name'],
      2                 columns = 'Date',
      3                 values=['Temperature'])

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:7876, in DataFrame.pivot(self, index, columns, values)
   7871 @Substitution("")
   7872 @Appender(_shared_docs["pivot"])
   7873 def pivot(self, index=None, columns=None, values=None) -> DataFrame:
   7874     from pandas.core.reshape.pivot import pivot
-> 7876     return pivot(self, index=index, columns=columns, values=values)

File ~\anaconda3\lib\site-packages\pandas\core\reshape\pivot.py:520, in pivot(data, index, columns, values)
    518     else:
    519         indexed = data._constructor_sliced(data[values]._values, index=multiindex)
--> 520 return indexed.unstack(columns_listlike)

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:8419, in DataFrame.unstack(self, level, fill_value)
   8357 """
```

```
   8358 Pivot a level of the (necessarily hierarchical) index labels.
   8359
   (...)
   8415 dtype: float64
   8416 """
   8417 from pandas.core.reshape.reshape import unstack
-> 8419 result = unstack(self, level, fill_value)
   8421 return result.__finalize__(self, method="unstack")

File ~\anaconda3\lib\site-packages\pandas\core\reshape\reshape.py:478, in unstack(obj, level, fill_value)
    476 if isinstance(obj, DataFrame):
    477     if isinstance(obj.index, MultiIndex):
--> 478         return _unstack_frame(obj, level, fill_value=fill_value)
    479     else:
    480         return obj.T.stack(dropna=False)

File ~\anaconda3\lib\site-packages\pandas\core\reshape\reshape.py:505, in _unstack_frame(obj, level, fill_value)
    503     return obj._constructor(mgr)
    504 else:
--> 505     unstacker = _Unstacker(obj.index, level=level, constructor=obj._constructor)
    506     return unstacker.get_result(
    507         obj._values, value_columns=obj.columns, fill_value=fill_value
    508     )

File ~\anaconda3\lib\site-packages\pandas\core\reshape\reshape.py:140, in _Unstacker.__init__(self, index, level, constructor)
    133 if num_cells > np.iinfo(np.int32).max:
    134     warnings.warn(
    135         f"The following operation may generate {num_cells} cells "
    136         f"in the resulting pandas object.",
    137         PerformanceWarning,
    138     )
--> 140 self._make_selectors()

File ~\anaconda3\lib\site-packages\pandas\core\reshape\reshape.py:192, in _Unstacker._make_selectors(self)
    189 mask.put(selector, True)
    191 if mask.sum() < len(self.index):
--> 192     raise ValueError("Index contains duplicate entries, cannot reshape")
    194 self.group_index = comp_index
    195 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 [187…
```
pd.pivot_table(data_tidy, index='Drug_Name', columns='Date', values=['Temperature'], agg
```

Out[187]:

	None		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 how we do 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 functions

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

In [188…
```
pd.pivot_table(data_tidy, index='Drug_Name', columns='Date', values=['Temperature', 'Pre
```

Out[188]:

None	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

Handling Missing Values

If you notive, there are many "NaN" values in our data

In [189…
```
data_tidy.head()
```

Out[189]:

None	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

What are these "NaN" values?

They are basically **missing values**

What are missing values?

A Missing Value signifies an **empty cell/no data**

There can be 2 kinds of missing values:

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

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

The diff mainly lies in their datatype

In [190…
```
type(None)
```

Out[190]:
```
NoneType
```

In [191…
```
type(np.nan)
```

Out[191]:
```
float
```

None type is for missing values in a column with **non-number entries**

- E.g.-strings

NaN occurs for columns with **number entries**

Note:

Pandas uses these values nearly **interchangeably**, converting between them where appropriate, based on column datatype

In [192…
```
pd.Series([1, np.nan, 2, None])
```

Out[192]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For **numerical** types, Pandas changes **None to NaN** type

In [193…
```
pd.Series(["1", "np.nan", "2", None])
```

Out[193]:
```
0         1
1    np.nan
2         2
3      None
dtype: object
```

In [194…
```
pd.Series(["1", "np.nan", "2", np.nan])
```

Out[194]:
```
0         1
1    np.nan
2         2
3       NaN
dtype: object
```

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

Now we have the basic idea about missing values

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

In [195…
```
data.isna().head()
```

Out[195]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30:00
0	False	False	False	False	False	True	False	False	False	False	False	False
1	False	False	False	False	False	True	False	False	False	False	False	False
2	False	False	False	True	False	False	True	False	False	True	True	False
3	False	False	False	True	False	False	True	False	False	True	True	False
4	False	False	False	False	True	True	False	True	False	False	False	False

We can also use isnull to get the same results

In [196…
```
data.isnull().head()
```

Out[196]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30:00
0	False	False	False	False	False	True	False	False	False	False	False	False
1	False	False	False	False	False	True	False	False	False	False	False	False
2	False	False	False	True	False	False	True	False	False	True	True	False
3	False	False	False	True	False	False	True	False	False	True	True	False
4	False	False	False	False	True	True	False	True	False	False	False	False

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

isnull() is just an alias for isna()

In [197…
```
pd.isnull
```

Out[197]:
```
<function pandas.core.dtypes.missing.isna(obj)>
```

In [198…
```
pd.isna
```

Out[198]:
```
<function pandas.core.dtypes.missing.isna(obj)>
```

As we can see, 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

In [199…
```
data.isna().sum()
```

Out[199]:
```
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

Can we also get the number of missing values in each row?

In [200…]
```
data.isna().sum(axis=1)
```

Out[200]:
```
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
```

In [201…]
```
data[data.isnull().any(axis = 1)]
```

Out[201]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	NaN	21.0	21.0	22	23.0	21.0	2
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	NaN	11.0	13.0	14	16.0	16.0	2
2	15-10-2020	docetaxel injection	Temperature	NaN	17.0	18.0	NaN	17.0	18	NaN	NaN	2
3	15-10-2020	docetaxel injection	Pressure	NaN	22.0	22.0	NaN	22.0	23	NaN	NaN	2
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	NaN	NaN	27.0	NaN	26	25.0	24.0	2
5	15-10-2020	ketamine hydrochloride	Pressure	8.0	NaN	NaN	7.0	NaN	9	10.0	11.0	1
6	16-	diltiazem	Temperature	34.0	35.0	36.0	36.0	37.0	38	37.0	38.0	3

	10-2020	hydrochloride										
7	16-10-2020	diltiazem hydrochloride	Pressure	18.0	19.0	20.0	21.0	22.0	23	24.0	25.0	2
8	16-10-2020	docetaxel injection	Temperature	46.0	47.0	NaN	48.0	48.0	49	50.0	52.0	5
9	16-10-2020	docetaxel injection	Pressure	23.0	24.0	NaN	25.0	26.0	27	28.0	29.0	2
10	16-10-2020	ketamine hydrochloride	Temperature	8.0	9.0	10.0	NaN	11.0	12	12.0	11.0	N
11	16-10-2020	ketamine hydrochloride	Pressure	12.0	12.0	13.0	NaN	15.0	15	15.0	15.0	N
12	17-10-2020	diltiazem hydrochloride	Temperature	20.0	19.0	19.0	18.0	17.0	16	15.0	NaN	1
13	17-10-2020	diltiazem hydrochloride	Pressure	3.0	4.0	4.0	4.0	6.0	8	9.0	NaN	

Note:

By default the value is `axis=0` in sum()

We 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

How can we drop rows containing null values?

In [202…
```
data.dropna()
```

Out[202]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30
14	17-10-2020	docetaxel injection	Temperature	12.0	13.0	14.0	15.0	16.0	17	18.0	19.0	2
15	17-10-2020	docetaxel injection	Pressure	20.0	22.0	22.0	22.0	22.0	23	25.0	26.0	2
16	17-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	16.0	17.0	18	19.0	20.0	2
17	17-10-	ketamine	Pressure	8.0	9.0	10.0	11.0	11.0	12	12.0	11.0	1

2020

Rows with **even a single missing value** have been deleted

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

In [203…
```
data.dropna(axis=1)
```

Out[203]:

	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

=> Every column which had even a single missing value has been deleted

But what are the problems with deleting rows/columns?

One of the major problems:

- loss of data

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

How can we fill the missing values with some data?

In [204…
```
data.fillna(0).head()
```

Out[204]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7:30:00	8:30:00	9:30:0
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	0.0	21.0	21.0	22	23.0	21.0	22

1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	0.0	11.0	13.0	14	16.0	16.0	24
2	15-10-2020	docetaxel injection	Temperature	0.0	17.0	18.0	0.0	17.0	18	0.0	0.0	23
3	15-10-2020	docetaxel injection	Pressure	0.0	22.0	22.0	0.0	22.0	23	0.0	0.0	27
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	0.0	0.0	27.0	0.0	26	25.0	24.0	23

What is fillna(0) doing?

It fills all missing values with 0

We can do the same on a particular column too

In [205…
```
data['2:30:00'].fillna(0)
```

Out[205]:
```
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
```

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

We can use some **kind of estimator** too

- An estimator like **mean or median**

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

In [206…
```
data['2:30:00'].mean()
```

Out[206]:
```
18.8125
```

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

In [207…
```
data['2:30:00'].fillna(data['2:30:00'].mean())
```

Out[207]:
```
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 a solution here?

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

In [208…
```
# data_tidy.groupby("Drug_Name")["Temperature"].mean()
```

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

We can use `apply` that we learnt earlier

Let's first create a function to calculate the mean

In [209…
```
def temp_mean(x):
  x['Temperature_avg'] = x['Temperature'].mean() # We will name the new col Temperature_
  return x
```

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

In [210…
```
data_tidy=data_tidy.groupby(["Drug_Name"]).apply(temp_mean)
data_tidy
```

Out[210]:

None	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 [211…
```
data_tidy['Temperature'].fillna(data_tidy["Temperature_avg"], inplace=True)
data_tidy
```

Out[211]:

None	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 [212…
```
data_tidy.isna().sum()
```

Out[212]:
```
None
Date                0
time                0
Drug_Name           0
Pressure           13
Temperature         0
Temperature_avg     0
dtype: int64
```

Great!!

We have removed the null values of our Temperature column

Let's do the same for Pressure

In [213…
```
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
```

Out[213]:

None	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 [214…

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

Out[214]:

```
None
Date                0
time                0
Drug_Name           0
Pressure            0
Temperature         0
Temperature_avg     0
Pressure_avg        0
dtype: int64
```

This gives us a **basic idea** about working with missing values

We will further learn more on this during later lectures of **feature engineering**

Pandas Cut

Sometimes, we would want our data to be in **categorical format instead of continous data**.

What do we mean by converting continous into categorical data?

Lets say, instead of knowing specific test values of a month, I want to know its type

What could be the types?

Depends on level of granularity we want to have - Low, Medium, High, V High

We could have defined more (or less) categories

But how can bucketisation of continous data help?

- Since, we can get the count of different categories
- We can get a idea of the bin which category (range of values) most of the temperature values lie.

What function can we use to convert cont. to cat. data?

- Will use pd.cut()
- We need to provide:
  - the continous data
  - bins edges (array of numbers) to "cut" the entire range
  - labels corresponding to every bin

Let's try to us this on our max (temp) column to categorise the data into bins

But, to define categories, lets first check min and max temp values

In [215…
```
data_tidy
```

Out[215]:

None	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 [216…
```
print(data_tidy['Temperature'].min(), data_tidy['Temperature'].max())
```

```
8.0 58.0
```

Min value = 8, Max value is 58.

- Lets's keep some buffer for future values and take the range from 5-60(instead of 8-58)
- Lets divide this data into 4 bins of 10-15 values each

In [217…
```
temp_points = [5, 20, 35, 50, 60]
temp_labels = ['low','medium','high','very_high'] # Here labels define the severity of t
data_tidy['temp_cat'] = pd.cut(data_tidy['Temperature'], bins=temp_points, labels=temp_l
data_tidy.head()
```

Out[217]:

None	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg	temp_cat
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242	low
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871	medium
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484	medium
3	15-10-	11:30:00	diltiazem	19.0	20.0	24.848485	15.424242	low

	2020		hydrochloride					
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871	medium

In [218…
```
data_tidy['temp_cat'].value_counts()
```

Out[218]:
```
low          50
medium       38
high         15
very_high     5
Name: temp_cat, dtype: int64
```

String function and motivation for datetime

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 "hydrochloric" in their drug name?

In [219…
```
data_tidy.loc[data_tidy['Drug_Name'].str.contains('hydrochloride')].head()
```

Out[219]:

None	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg	temp_cat
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242	low
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484	medium
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242	low
5	15-10-2020	11:30:00	ketamine hydrochloride	9.0	21.0	17.709677	11.935484	medium
6	15-10-2020	12:30:00	diltiazem hydrochloride	20.0	21.0	24.848485	15.424242	medium

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 [220…
```
data_tidy['Date'].str.split('-')
```

Out[220]:
```
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 [221…
```
data_tidy['Date'].str.split('-').apply(lambda x:x[2])
```

Out[221]:
```
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 of pandas

Datetime

Lets start with understanding a date-time type of data

How can we handle handle date-time data-types?

- We can do using 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

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

In [222…
```
data_tidy['timestamp'] = data_tidy['Date']+ " "+ data_tidy['time']
```

In [223…
```
data_tidy.drop(['Date', 'time'], axis=1, inplace=True)
```

In [224…
```
data_tidy.head()
```

Out[224]:

None	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg	temp_cat	timestamp
0	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242	low	15-10-2020 10:30:00
1	docetaxel injection	26.0	23.0	30.387097	25.483871	medium	15-10-2020 10:30:00

2	ketamine hydrochloride	9.0	22.0	17.709677	11.935484	medium	15-10-2020 10:30:00
3	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242	low	15-10-2020 11:30:00
4	docetaxel injection	29.0	25.0	30.387097	25.483871	medium	15-10-2020 11:30:00

Lets convert our `timestamp` col now

In [225…
```
data_tidy['timestamp'] = pd.to_datetime(data_tidy['timestamp']) # will leave to explore

data_tidy
```

Out[225]:

None	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg	temp_cat	timestamp
0	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242	low	2020-10-15 10:30:00
1	docetaxel injection	26.0	23.0	30.387097	25.483871	medium	2020-10-15 10:30:00
2	ketamine hydrochloride	9.0	22.0	17.709677	11.935484	medium	2020-10-15 10:30:00
3	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242	low	2020-10-15 11:30:00
4	docetaxel injection	29.0	25.0	30.387097	25.483871	medium	2020-10-15 11:30:00
...	...	...	...	...	...	...	...
103	docetaxel injection	26.0	19.0	30.387097	25.483871	low	2020-10-17 08:30:00
104	ketamine hydrochloride	11.0	20.0	17.709677	11.935484	low	2020-10-17 08:30:00
105	diltiazem hydrochloride	9.0	13.0	24.848485	15.424242	low	2020-10-17 09:30:00
106	docetaxel injection	27.0	20.0	30.387097	25.483871	low	2020-10-17 09:30:00
107	ketamine hydrochloride	12.0	21.0	17.709677	11.935484	medium	2020-10-17 09:30:00

108 rows × 7 columns

In [226…
```
data_tidy.info()
```

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

```
dtypes: category(1), datetime64[ns](1), float64(4), object(1)
memory usage: 10.3+ KB
```

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

Now, Let's look at a single timestamp using Pandas

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

In [227…
```
ts = data_tidy['timestamp'][0]
ts
```

Out[227]:
```
Timestamp('2020-10-15 10:30:00')
```

Now how can we extract the year from this date ?

In [228…
```
ts.year
```

Out[228]:
```
2020
```

Similarly we can also access the month and day using the `month` and `day` attributes

In [229…
```
ts.month
```

Out[229]:
```
10
```

In [230…
```
ts.day
```

Out[230]:
```
15
```

But what if we want to know the name of the month or the day of the week on that date ?

- We can find it using `month_name()` and `day_name()` methods

In [231…
```
ts.month_name()
```

Out[231]:
```
'October'
```

In [232…
```
ts.day_name()
```

Out[232]:
```
'Thursday'
```

In [233…
```
ts.dayofweek
```

Out[233]:
```
3
```

In [234…
```
ts.hour
```

Out[234]:
```
10
```

In [235…
```
ts.minute
```

Out[235]:
```
30
```

... and so on

We can similarly extract minutes and seconds

This data parsing from string to date-time makes it easier to work with data

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

In [236…
```
data_tidy['timestamp'].dt
```

Out[236]:
```
<pandas.core.indexes.accessors.DatetimeProperties object at 0x000001A2D13C7460>
```

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

In [237…
```
data_tidy['timestamp'].dt.year
```

Out[237]:
```
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
```

Now, Let's **create the new column using these extracted values from the property**

We will use strfttime, short for stringformat time, to modify our datetime format

Let's learn this with the help of few examples

In [238…
```
data_tidy['timestamp'][0]
```

Out[238]:
```
Timestamp('2020-10-15 10:30:00')
```

In [239…
```
print(data_tidy['timestamp'][0].strftime('%Y')) # Formatter for year
```

```
2020
```

In [240…
```
print(data_tidy['timestamp'][0].strftime('%m')) # Formatter for month
```

```
10
```

In [241…
```
print(data_tidy['timestamp'][0].strftime('%d')) # Formatter for day
```

```
15
```

In [242…
```
print(data_tidy['timestamp'][0].strftime('%H')) # Formatter for hour
```

```
10
```

In [243…
```
print(data_tidy['timestamp'][0].strftime('%M')) # Formatter for minutes
```

```
30
```

In [244…
```
print(data_tidy['timestamp'][0].strftime('%S')) # Formatter for seconds
```

```
00
```

Similarly we can combine the format types to modify the date-time format as per our convinience

In [245…
```
data_tidy['timestamp'][0].strftime('%m-%d')
```

Out[245]:
```
'10-15'
```

Writing to file

How can we write our dataframe to a csv file?

- We have to **provide the path and file_name** in which you want to store the data

In [246…
```
data_tidy.to_csv('pfizer_tidy.csv', sep=",")
```

To find all the values from the series that starts with a pattern "s":

SQL - WHERE column_name LIKE 's%'

Python - column_name.str.startswith('s')

To find all the values from the series that ends with a pattern "s":

SQL - WHERE column_name LIKE '%s'

Python - column_name.str.endswith('s')

To find all the values from the series that contains pattern "s":

SQL - WHERE column_name LIKE '%s%'

Python - column_name.str.contains('s')

Thank You!

In [ ]:
```
```