Table of contents

Pandas

Time to learn about the most important `data analysis/data processing` tool in the world.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Installation

If you have `conda` installed, Just run the following command in your terminal:

```
conda install pandas
```

Or if you dont't have `conda` installed, just run `pip install pandas` in your terminal.

`Pandas` is a `python` library for `data analysis`. Some say it's python's version of `excel`. Well they are not wrong but I would say it's more powerful and easy to use than `excel` and has a lot of features that is way easier to use than `excel`.

For getting the grasp of `pandas` you need to know `two` crucial concepts:

1. `Series`
2. `DataFrame`

And also `pandas` is built on top of `numpy`. So, you need general understanding of `numpy` to understand `pandas` better.

> In my github repo I have section dedicated to `numpy` if you want to learn more about it.

So, now let's talk about `pandas` series.

Series

A `Series` is a `one-dimensional` array-like object. It is similar to `numpy` array.

I'll make 2 variables:

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

data = np.array([5,2,4,2])
labels = ['a','b','c','d']
```

I made a simple `numpy` array called `data` and another list variable called `labels`.

Now, I use the `data` variable to create a `pandas series`.

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

series = pd.Series(data)
series
```

Out[2]:
```
0    5
1    2
2    4
3    2
dtype: int64
```

And would you look at that!

I used the `pandas.Series` to create a `Series` from the `data` array and it looks like a table with `indexes`. Each index corresponds to a value of the array.

This is what a `Series` is.

Now, what we can do is use these `indexes` to get the a value from the `Series`.

In [3]:
```
series[0]
```

Out[3]:
```
np.int64(5)
```

See? So, this a series is a pandas representation of a `numpy` array. Right?

Looks like that.

But, no event though it looks like that, it's more than that.

In a normal pandas array, the indexes are unchangeable and always start from `0`. But in a series we can set custom indexes.

That's why I tool another python list called `labels` and now what I can do is set the indices of the series to the values of `labels`.

In [4]:
```python
series2 = pd.Series(data, index=labels)
series2
```

Out[4]:
```
a    5
b    2
c    4
d    2
dtype: int64
```

And now we have a pandas series with custom index. Hmm.... why does this sound familiar? Did we do something like this in python?

Yes, we do have a list is python where we can do custom indexing. And that is a `dictionary` or `hashmap` in python. So, we can say that instead of being pandas version of a `numpy` array, it is more like a `pandas` verison of a `python` `dictionary`.

> A `dictionary` is a `hashmap` in `python`. A hashmap is a data structure that maps `keys` to `values`. Key are like custom index and values are the corresponding values to the keys.

We can also set the index of a `pandas` series that doesn't have a custom index. Like the `series` variable in the previous code block.

In [5]:
```python
series.index = labels #setting the index
series
```

Out[5]:
```
a    5
b    2
c    4
d    2
dtype: int64
```

> pandas series gives us a attributes like `index` and we can assign values to it. And in the code block above, I assigned the labels variable as the index of the `series`.

There is another way we can make a pandas `series`.

In [6]:
```python
data = {'a': 1, 'b': 2, 'c': 3}

series3 = pd.Series(data)

series3
```

Out[6]:
```
a    1
b    2
c    3
dtype: int64
```

That wasn't a shocker, was it? We can make a `pandas` series by directly passing a python `dictionary` to `pandas.Series` function. The `keys` are the `index` and the `values` are the `values`.

In the long run we will not be using `serieses` that much for data manipulation. But series is a building block for the most important thing in `pandas` which is `dataframes`.

So, It is impoertant we understand `series` to have a better grasp of `pandas`.

Dataframes

basics

Simly, `DataFrames` are `tables` of data.

I think showing you would be more effective.

In [7]:
```python
np.random.seed(1)

data = np.random.randn(6,5)
data
```

Out[7]:
```
array([[ 1.62434536, -0.61175641, -0.52817175, -1.07296862,  0.86540763],
       [-2.3015387 ,  1.74481176, -0.7612069 ,  0.3190391 , -0.24937038],
       [ 1.46210794, -2.06014071, -0.3224172 , -0.38405435,  1.13376944],
       [-1.09989127, -0.17242821, -0.87785842,  0.04221375,  0.58281521],
       [-1.10061918,  1.14472371,  0.90159072,  0.50249434,  0.90085595],
       [-0.68372786, -0.12289023, -0.93576943, -0.26788808,  0.5303554
7]])
```

> Seed() is a mathod to freeze the random number generator in a particular random state so that the same random numbers are generated again and again.

I made a `2D` array called `data` with 6 rows and 5 columns.

Now, we can make a `pandas` dataframe from the `data` array.

`DataFrame()` method works almost exactly like `Series()` method. But the only change is that it has a extra argument. A table has rows and columns. So, to represent the rows and columns We have to pass them as arguments to the `DataFrame()` method.

```
In [8]: df = pd.DataFrame(
            data=data,
            index=['row1', 'row2', 'row3', 'row4', 'row5', 'row6'],
            columns=['col1', 'col2', 'col3', 'col4', 'col5']
        )
        df
```

Out[8]:

	col1	col2	col3	col4	col5
row1	1.624345	-0.611756	-0.528172	-1.072969	0.865408
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

In one command we have a `pandas` dataframe. With the specified `index` and `columns`.

And we can select any column if we want.

```
In [9]: df['col1']
```

```
Out[9]: row1    1.624345
        row2   -2.301539
        row3    1.462108
        row4   -1.099891
        row5   -1.100619
        row6   -0.683728
        Name: col1, dtype: float64
```

```
In [10]: df['col2']
```

```
Out[10]: row1   -0.611756
         row2    1.744812
         row3   -2.060141
         row4   -0.172428
         row5    1.144724
         row6   -0.122890
         Name: col2, dtype: float64
```

Have we see this kinda output before?

If we check out the type of a single column, we might get a surprise.

```
In [11]: type(df['col1'])
```

```
Out[11]: pandas.core.series.Series
```

Owh!

That's surprising.

It's a `pandas` series.

Does that mean every single column in a `pandas` dataframe is a `pandas` series?

`YES!`

Pandas `dataframe` is a collection of multiple `pandas` series.

If we see type of the whole `dataframe`.

In [12]:
```
type(df)
```

Out[12]:
```
pandas.core.frame.DataFrame
```

We can clearly see that it is a `pandas` dataframe. But individual columns are `pandas` series.

And that means the indexes are `series` indexes that are common to all the columns.

It's an important detail that will come in handy later on.

Now, what about selecting multiple columns?

We can do that by passing a list inside the `square brackets` to the `pandas` dataframe.

In [13]:
```
df[['col2', 'col3', 'col3']]
```

Out[13]:

	col2	col3	col3
row1	-0.611756	-0.528172	-0.528172
row2	1.744812	-0.761207	-0.761207
row3	-2.060141	-0.322417	-0.322417
row4	-0.172428	-0.877858	-0.877858
row5	1.144724	0.901591	0.901591
row6	-0.122890	-0.935769	-0.935769

It'll give us the data with the columns we need.

And we can also make new columns if we want.

In [14]:
```
new_column_data = np.random.randn(3)
new_column_data
```

Out[14]:
```
array([-0.69166075, -0.39675353, -0.6871727 ])
```

I generated a `numpy` array called `new_column_data` with 3 elements.

Now we can add these as new column by:

In [15]:

```
df['new'] = new_column_data
df
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[15], line 1
----> 1 df['new'] = new_column_data
      2 df

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/fram
e.py:4322, in DataFrame.__setitem__(self, key, value)
   4319     self._setitem_array([key], value)
   4320 else:
   4321     # set column
-> 4322     self._set_item(key, value)

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/fram
e.py:4535, in DataFrame._set_item(self, key, value)
   4525 def _set_item(self, key, value) -> None:
   4526     """
   4527     Add series to DataFrame in specified column.
   4528 
   (...)   4533     ensure homogeneity.
   4534     """
-> 4535     value, refs = self._sanitize_column(value)
   4537     if (
   4538         key in self.columns
   4539         and value.ndim == 1
   4540         and not isinstance(value.dtype, ExtensionDtype)
   4541     ):
   4542         # broadcast across multiple columns if necessary
   4543         if not self.columns.is_unique or isinstance(self.columns, M
ultiIndex):

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/fram
e.py:5288, in DataFrame._sanitize_column(self, value)
   5285     return _reindex_for_setitem(value, self.index)
   5287 if is_list_like(value):
-> 5288     com.require_length_match(value, self.index)
   5289 arr = sanitize_array(value, self.index, copy=True, allow_2d=True)
   5290 if (
   5291     isinstance(value, Index)
   5292     and value.dtype == "object"
   (...)   5295     # TODO: Remove kludge in sanitize_array for string mode
when enforcing
   5296     # this deprecation

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/comm
on.py:573, in require_length_match(data, index)
    569 """
    570 Check the length of data matches the length of the index.
    571 """
    572 if len(data) != len(index):
--> 573     raise ValueError(
    574         "Length of values "
    575         f"({len(data)}) "
    576         "does not match length of index "
    577         f"({len(index)})"
    578     )

ValueError: Length of values (3) does not match length of index (6)
```

ANNNND we have a big big error.

This error is saying that the new column does not have enough rows.

> If you want to make a new column you have to make sure it has the same amount of rows as the dataframe.

As I made a column with only 3 random values, where the dataframe has `6` rows, we get this error.

So, we make a `column` with the same number of rows as the `dataframe` and then we add it to the `dataframe`.

In [16]:
```
new_column_data = np.random.randn(6)
df['new'] = new_column_data
df
```

Out[16]:

	col1	col2	col3	col4	col5	new
row1	1.624345	-0.611756	-0.528172	-1.072969	0.865408	-0.845206
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370	-0.671246
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769	-0.012665
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815	-1.117310
row5	-1.100619	1.144724	0.901591	0.502494	0.900856	0.234416
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355	1.659802

And it is done.

We have a new column called `new` in the `dataframe`.

Now, to `remove` a column we can simply do:

In [17]:
```
df.drop(columns='new')
```

Out[17]:

	col1	col2	col3	col4	col5
row1	1.624345	-0.611756	-0.528172	-1.072969	0.865408
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

And done!

`The new column is gone! or is it?!`

In [18]:
```
df
```

Out[18]:

	col1	col2	col3	col4	col5	new
row1	1.624345	-0.611756	-0.528172	-1.072969	0.865408	-0.845206
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370	-0.671246
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769	-0.012665
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815	-1.117310
row5	-1.100619	1.144724	0.901591	0.502494	0.900856	0.234416
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355	1.659802

Huh!

Why is the column still there even though we `removed` it? in the previous command?

> The `drop()` method returns a new `pandas` dataframe that does not have the column we removed. But it doesn't permanently remove the column.

And that's why to permanently remove a column we can pass an extra argument called `inplace=True` to the `drop()` method.

This tells the `drop()` method to permanently remove the column.

In [19]:

```
df.drop(
    columns='new',
    inplace=True
)
df
```

Out[19]:

	col1	col2	col3	col4	col5
row1	1.624345	-0.611756	-0.528172	-1.072969	0.865408
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

And the new column is `permanently` removed.

Now, you might ask can we do the same for a row?

Yes, we can.

It's exactly the same way.

But you need to know about `axis` argument.

In pandas, the columns and rows are given a `axis` for better indexing and selection.

> `axis 1` is for the columns and `axis 0` is for the rows.

In the drop method the axis is set to `1` by default. So, if we just say `drop('row1', inplace=True)` it will should show an error and it used to show and error on the previous versions of pandas but now it doesn't.

We can directly just pass the name of the `row` and it'll remove the row.

In [20]:
```
df.drop('row1')
```

Out[20]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

Automatically finds if it's a row or a column.

But it is bets practice to use the `axis` argument.

In [21]:
```
df.drop(
    'row1',
    axis=0,
    inplace=True
)

df
```

Out[21]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

Now, I told you before that `pandas` is built upon `numpy` and `numpy` is the main driving force and pandas is just a tool to make our task easier for numpy arrays.

So, all the `pandas` data structures are just `numpy` arrays with index markers and some methods to make our life easier.

In [22]:
```
df.shape
```

Out[22]:
```
(5, 5)
```

Now, as you can see our `dataframe` is still a `2d numpy` array with shape (5,5) because we removed a row.

So, as it is a numpy 2d array we should be able to do indexing and selecting just like we do in numpy.

And we can.

Let's say I want all the values of `row 3`.

In [23]:
```
df.loc['row3']
```

Out[23]:
```
col1    1.462108
col2   -2.060141
col3   -0.322417
col4   -0.384054
col5    1.133769
Name: row3, dtype: float64
```

> We have to use the `loc or location` method to select a row individually.

And after that we can do `indexing` just like we do in `numpy` arrays.

In the above code block I selected the 3rd row and it looks awfully like a `pandas` series. Is it a pandas series?

In [24]:
```
type(df.loc['row3'])
```

Out[24]:
```
pandas.core.series.Series
```

`YES!` it is.

Every indivisual row and column in a `pandas` dataframe is a `pandas series`.

Now, what about a `specific` value selection? Like I want to see the value of 3rd row and 4th column?

Just ike we did in numpy arrays. We can get a value from a `pandas` dataframe.

In [25]:
```
df.loc['row3', 'col4']
```

Out[25]:
```
np.float64(-0.38405435466841564)
```

And the fun part is we select multiple rows and column at the same time.

Just like we can select multiple columns by passing a `list` to the `square brackets` to the `pandas` dataframe.

We can do the same with the `loc` method.

In [26]:
```
df.loc[['row3', 'row5'], ['col4', 'col5']]
```

Out[26]:

	col4	col5
row3	-0.384054	1.133769
row5	0.502494	0.900856

It'll give us a `dataframe` with 2 rows and 2 columns.

And also another thing we need to discuss is that in pandas, it also gives a way to select a rows by index. Even though we set custom rows. We can use a special `method` for that.

> The method is called `iloc` or `integer location` method.

In [27]:
```
df.iloc[3]
```

Out[27]:
```
col1   -1.100619
col2    1.144724
col3    0.901591
col4    0.502494
col5    0.900856
Name: row5, dtype: float64
```

Returns the row with index 3, which is `row5`.

In [28]:
```
df.iloc[3, 4]
```

Out[28]:
```
np.float64(0.9008559492644118)
```

This returns the value of the row with index 3 and column with index 4. So, index 3 is `row5` and index 4 is `col5`.

In [29]:
```
df.loc['row5', 'col5']
```

Out[29]:
```
np.float64(0.9008559492644118)
```

As you can see exactly the same thing.

There are a lot of different ways to select/index in `pandas`. But in the end it is a `2d numpy` array with custom index `markers`.

And to be honest we will not be using pandas like this. Most of the time we will be using pandas for data `manipulation`. Which is why you need to learn about `conditional selection` in `pandas`.

Conditional Selection

In the `numpy` article I talked about `conditional selection` in `numpy` arrays. If you haven't read that article then please read it first.

`Conditional` selection in `pandas` and in `data analysis` is a very important concept.

For example. Let's say I want to select all the elements that are greater than `0`.

In [30]:
```
df[df > 0]
```

Out[30]:

	col1	col2	col3	col4	col5
row2	NaN	1.744812	NaN	0.319039	NaN
row3	1.462108	NaN	NaN	NaN	1.133769
row4	NaN	NaN	NaN	0.042214	0.582815
row5	NaN	1.144724	0.901591	0.502494	0.900856
row6	NaN	NaN	NaN	NaN	0.530355

Let's me break it down what just happened.

First,

In [31]:
```
df > 0
```

Out[31]:

	col1	col2	col3	col4	col5
row2	False	True	False	True	False
row3	True	False	False	False	True
row4	False	False	False	True	True
row5	False	True	True	True	True
row6	False	False	False	False	True

I'm Simply checking `df > 0` and it returns a `dataframe` with `True` and `False` values.

These boolean values correspond to the original dataframe when the condition is met.

We can see that the value of `row2` and `col2` is `True` that means the value of the original dataframe should be grater then 0.

In [32]:
```
df.loc['row2', 'col2']
```

Out[32]:
```
np.float64(1.74481176421648)
```

And yes it is greater than 0.

Now, we can pass this `boolean` dataframe as a indexing argument to the `pandas` dataframe and get a filtered dataframe Where only the true values are shown and the false values are set to `NaN` values.

In [33]:
```
boolDF = df > 0
boolDF
```

Out[33]:

	col1	col2	col3	col4	col5
row2	False	True	False	True	False
row3	True	False	False	False	True
row4	False	False	False	True	True
row5	False	True	True	True	True
row6	False	False	False	False	True

In [34]:

```
df[boolDF]
```

Out[34]:

	col1	col2	col3	col4	col5
row2	NaN	1.744812	NaN	0.319039	NaN
row3	1.462108	NaN	NaN	NaN	1.133769
row4	NaN	NaN	NaN	0.042214	0.582815
row5	NaN	1.144724	0.901591	0.502494	0.900856
row6	NaN	NaN	NaN	NaN	0.530355

> It is good practice to break down the conditional selection into multiple steps for better understanding and readability.

In a dataframe we can do this conditional selection on a single column.

Let's see what happens if we apply the same condition in `col4`.

In [35]:

```
df['col4'] > 0
```

Out[35]:

```
row2     True
row3    False
row4     True
row5     True
row6    False
Name: col4, dtype: bool
```

And now it returns a series with `True` and `False` values for every row.

I'll just store this series in a variable and apply this condition to the whole dataframe.

In [36]:

```
boolSR = df['col4'] > 0
boolSR
```

Out[36]:

```
row2     True
row3    False
row4     True
row5     True
row6    False
Name: col4, dtype: bool
```

In [37]:

```
df[boolSR]
```

Out[37]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856

And we have a `filtered` dataframe.

Conditional selection on a single column gives us a filtered dataframe, where we can see that the `values` that were true for `col4` are now visible along with the `values` of the other columns even if they don't meet the condition.

It's like saying give me the dataframe where `col4` is greater than 0.

We can do the same thing, like saying give me the dataframe where `col5` is less than 0.

In [38]:
```
boolSR = df['col5'] < 0
df[boolSR]
```

Out[38]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.24937

And we can see, there's only one row where `col5` is less than 0.

Now, as this is a `dataframe` If we want we can store it in a variable and do conditional selection or indexing and everything we can do in a `dataframe`.

Like let's say I only want the `col2` and `col4` columns where `col3` is greater than 0.

In [39]:
```
boolSR = df['col4'] > 0 # Boolean Series
boolSR
```

Out[39]:
```
row2     True
row3    False
row4     True
row5     True
row6    False
Name: col4, dtype: bool
```

In [40]:
```
res_df = df[boolSR]
res_df
```

Out[40]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856

In [41]:
```
res_df[['col2', 'col4']]
```

Out[41]:

	col2	col4
row2	1.744812	0.319039
row4	-0.172428	0.042214
row5	1.144724	0.502494

As you can see the we can easily do conditional selection if we break it down into multiple steps.

And if you practice enough you'll eventually be able to do the same operations in 1 line.

In [42]:
```
df[df['col4']>0][['col2', 'col4']]
```

Out[42]:

	col2	col4
row2	1.744812	0.319039
row4	-0.172428	0.042214
row5	1.144724	0.502494

> It's exactly the same thing as breaking the conditional selection into multiple steps.

One more very important thing I want to talk about is `multiple conditional selection` in `pandas` dataframes.

I gave some examples of how we can filter out data using a single condition on a `column` but what if we want to filter out data using multiple conditions on a single or multiple columns?

It's common python coding practice to use `and` and `or` operators to combine multiple conditions. They are called `logical operators` in python.

So, let's find the rows where `col3` is greater than 0 `or` `col4` is greater than 0.

In [43]:
```
boolSR = (df['col3'] > 0) or (df['col4'] > 0)
boolSR
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_10158/4148211646.py in ?()
----> 1 boolSR = (df['col3'] > 0) or (df['col4'] > 0)
      2 boolSR

~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/generic.p
y in ?(self)
   1578     @final
   1579     def __nonzero__(self) -> NoReturn:
-> 1580         raise ValueError(
   1581             f"The truth value of a {type(self).__name__} is ambiguo
us. "
   1582             "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
   1583         )

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool
(), a.item(), a.any() or a.all().
```

and it's throughing an error. And this error tells us that `The truth value of a Series is ambiguous` which means that when we are using `or` operator it cannot decide between `True` and `False` values because the `logical` operators are used to only 2 boolean values not a whole series.

So, to fix this issue we can use the `bitwise or` operator `|` instead of `logical or` operator.

Same standard goes for `and` operator.

So, to combine multiple conditional selections we can use `|` and `&` operators for `or` and `and` respectively.

In [44]:
```
boolSR = (df['col3'] > 0) | (df['col4'] > 0)
boolSR
```

Out[44]:
```
row2     True
row3    False
row4     True
row5     True
row6    False
dtype: bool
```

> AND ALWAYS REMEMBER TO USE `()` around the conditions.

Now, we have a `combined` boolean series. And we can use this boolean series to filter out the dataframe.

In [45]:
```
res = df[boolSR]
res
```

Out[45]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856

And that's how you combine multiple conditional selections in `pandas` dataframes.

And before we go any further I want to show you one more thing.

Resetting the index

In this article we have been using the same `dataframe` `df`. And I set it's index and columns manually.

Now, when your rows get very large it might be a little bit hard to manually set the index and columns.

And in that case you can completely reset the index by doing the following.

In [46]:
```
df.reset_index()
```

Out[46]:

	index	col1	col2	col3	col4	col5
0	row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
1	row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
2	row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
3	row5	-1.100619	1.144724	0.901591	0.502494	0.900856
4	row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

And if take a good look at the `dataframe` that we get. It has a new `index` starting from `0` and the old index is now a new column of the dataframe.

This is not inplace so the original `dataframe` is not changed.

In [47]:
```
df
```

Out[47]:

	col1	col2	col3	col4	col5
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815
row5	-1.100619	1.144724	0.901591	0.502494	0.900856
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

We can set the `inplace` argument to `True` to make the changes inplace/permanently. Also, you might not want the old index to be a column so you can pass `drop=True` to the `reset_index()` method.

In [48]:
```
df.reset_index(
    drop=True,
    # inplace=True
)
```

Out[48]:

	col1	col2	col3	col4	col5
0	-2.301539	1.744812	-0.761207	0.319039	-0.249370
1	1.462108	-2.060141	-0.322417	-0.384054	1.133769
2	-1.099891	-0.172428	-0.877858	0.042214	0.582815
3	-1.100619	1.144724	0.901591	0.502494	0.900856
4	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

And the old index is now removed from the `dataframe`.

> I didn't do it inplace because I want to use this `df` for more examples later.

Another thing we can do is set an `existing` column as the index.

For that Let me make a custom `column` again.

In [49]:
```
new = ['custom1', 'custom2', 'custom3', 'custom4', 'custom5']
```

Now we acn add this as a new column for our dataframe.

In [50]:
```
df['new'] = new
df
```

Out[50]:

	col1	col2	col3	col4	col5	new
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370	custom1
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769	custom2
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815	custom3
row5	-1.100619	1.144724	0.901591	0.502494	0.900856	custom4
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355	custom5

In [51]:
```
df
```

Out[51]:

	col1	col2	col3	col4	col5	new
row2	-2.301539	1.744812	-0.761207	0.319039	-0.249370	custom1
row3	1.462108	-2.060141	-0.322417	-0.384054	1.133769	custom2
row4	-1.099891	-0.172428	-0.877858	0.042214	0.582815	custom3
row5	-1.100619	1.144724	0.901591	0.502494	0.900856	custom4
row6	-0.683728	-0.122890	-0.935769	-0.267888	0.530355	custom5

Now we can make this new column as the index using the `set_index` method.

In [52]:

```
df.set_index(
    'new',
)
```

Out[52]:

	col1	col2	col3	col4	col5
new					
custom1	-2.301539	1.744812	-0.761207	0.319039	-0.249370
custom2	1.462108	-2.060141	-0.322417	-0.384054	1.133769
custom3	-1.099891	-0.172428	-0.877858	0.042214	0.582815
custom4	-1.100619	1.144724	0.901591	0.502494	0.900856
custom5	-0.683728	-0.122890	-0.935769	-0.267888	0.530355

And the new column is set as a new index and the older index is removed.

`set_index()` mathod has both `drop` and `inplace` arguments just like `reset_index()` method.

> Drop is set to `True` by default. inplace is set to `False` by default.

And this is how we can `reset` and `set` the index of a `pandas` dataframe.

Multilevel Indexing

Well, This is the boring part that I thought I was going to skip. But as this is called advanced indexing it is important to know about.

`Multilevel/hierarchical` indexing is when we have multiple levels of index in a `pandas` dataframe.

For example, let's say we have a `dataframe` with 2 rows and each row has 3 rows inside them and the whole data set has 2 columns.

In [53]:

```
groups = ['group1']*3 + ['group2']*3
inside_groups = ['row1', 'row2', 'row3']*2
columns = ['col1', 'col2']

tup = list(zip(groups, inside_groups))
```

In [54]:
```
tup
```

Out[54]:
```
[('group1', 'row1'),
 ('group1', 'row2'),
 ('group1', 'row3'),
 ('group2', 'row1'),
 ('group2', 'row2'),
 ('group2', 'row3')]
```

We have to make a tuple representation of the table. You can see that we have 2 groups and each has 3 rows.

And now let's use that to make a `pandas` dataframe.

In [55]:
```
index = pd.MultiIndex.from_tuples(
    tuples=tup,
)
index
```

Out[55]:
```
MultiIndex([('group1', 'row1'),
            ('group1', 'row2'),
            ('group1', 'row3'),
            ('group2', 'row1'),
            ('group2', 'row2'),
            ('group2', 'row3')],
           )
```

`pd.MultiIndex.from_tuples()` method will create a `heierarchical` index that we can pass as the `index` argument to the `DataFrame()` method.

In [56]:
```
df = pd.DataFrame(
    data=np.random.randn(6, 2),
    index=index,
    columns=columns,
)

df
```

Out[56]:

		col1	col2
group1	**row1**	0.742044	-0.191836
	row2	-0.887629	-0.747158
	row3	1.692455	0.050808
group2	**row1**	-0.636996	0.190915
	row2	2.100255	0.120159
	row3	0.617203	0.300170

And we have a `pandas` dataframe with a hierarchical index.

Now how do we `get data` or `slice` or other things from this dataframe?

Almost the same as before.

In [57]:
```
df.loc['group1']
```

Out[57]:

	col1	col2
row1	0.742044	-0.191836
row2	-0.887629	-0.747158
row3	1.692455	0.050808

We can use the `loc` method to select a group and it'll return a `dataframe` of that group.

Now, if we want to get specific data, like `group2`, `row2`:

In [58]:
```
group2 = df.loc['group2']
group2
```

Out[58]:

	col1	col2
row1	-0.636996	0.190915
row2	2.100255	0.120159
row3	0.617203	0.300170

In [59]:
```
group2.loc['row2']
```

Out[59]:
```
col1    2.100255
col2    0.120159
Name: row2, dtype: float64
```

Or we can directly index.

In [60]:
```
df.loc['group2'].loc['row2']
```

Out[60]:
```
col1    2.100255
col2    0.120159
Name: row2, dtype: float64
```

You just have to remember which `level` you are on. We can think of this as `groups` as `level 1` and `rows` as `level 2`.

So, we can use the `loc` method to select a group and then use the `loc` method agian to select a `row` from that group.

For better readability we can name the groups and rows.

In [61]:
```
df.index.names
```

Out[61]:
```
FrozenList([None, None])
```

This shows us that we haven't named the indices. So, we can name them. Let's name them `groups` and `rows`.

In [62]:
```
df.index.names = ['groups', 'rows']
df
```

Out[62]:

groups	rows	col1	col2
group1	row1	0.742044	-0.191836
	row2	-0.887629	-0.747158
	row3	1.692455	0.050808
group2	row1	-0.636996	0.190915
	row2	2.100255	0.120159
	row3	0.617203	0.300170

And we have names for the indices.

This opens up a whole new world of advanced indexing.

And that it the `cross seciton` or `xs()` method for a `pandas` dataframe.

Let's say we want `group2`

In [63]:
```
df.loc['group2']
```

Out[63]:

rows	col1	col2
row1	-0.636996	0.190915
row2	2.100255	0.120159
row3	0.617203	0.300170

We can simply do it with `loc`. But when we are working with a `hierarchical` dataframe we can use the `xs()` method to do the same thing.

> `xs()` is a method not a indexing attribute like `loc`.

In [64]:
```
df.xs('group2')
```

Out[64]:

rows	col1	col2
row1	-0.636996	0.190915
row2	2.100255	0.120159
row3	0.617203	0.300170

And it works. But one thing this method gives us is the power to skip levels.

Like if we want `row2` from noth groups.

In [65]:
```
df.xs(
    'row2',
    level='rows',
)
```

Out[65]:

	col1	col2
groups		
group1	-0.887629	-0.747158
group2	2.100255	0.120159

And we can do it by passing `rows2` and `level='rows'` as arguments to the `xs()` method and it'll skip the `groups` level and return a dataframe of `row2` only.

> That's why we can see group1 and group2 as indexes in the dataframe.

That's the gist of pandas dataframes.

Pandas `DataFrames` will stay with you for a very long long long time. So, getting a better understanding of indexing and Selection along with slicing and manipulating data is very important.

And one of the most important part of data handling is `missing` data handling. So, that's the next part of the series.

Handling Missing Data

In almost all `data sets` there will be missing data. And sometimes these missing data can cause `machine learning models` to not perform well.

So, we need a way to handle missing data.

Now, with different data sets, you might need to do different things to handle missing data effectively. In this section I'll be showing you how we can handle missing data `in general`.

So, let's make a dataframe with some missign data.

In [66]:

```
data = {
    'A': [1, 2, np.nan, 3],
    'B': [2, np.nan, np.nan, 4],
    'C': [7, 5, 6, 8]
}

df = pd.DataFrame(data)
df
```

Out[66]:

	A	B	C
0	1.0	2.0	7
1	2.0	NaN	5
2	NaN	NaN	6
3	3.0	4.0	8

Now, we have a `pandas` dataframe with some missing data. And if we want to remove all the rows with missing data we can use the `dropna()` method.

> `dropna()` will drop all the rows with missing data.

DropNa

In [67]:
```
df.dropna()
```

Out[67]:

	A	B	C
0	1.0	2.0	7
3	3.0	4.0	8

> The columns with the missing data will be transformed to a `float` data type because pandas will try to keep as much information as possible for a column with missing data.

> Column `C` if not a `float` data type because it didn't have any missing data.

`dropna()` will check for missing values in every row and if that row has any missing values it will drop that row immediately. Now, that is not permanent, because `dropna()` also has a `inplace` argument which is set to `false` by default.

Now, there's also another argument called `axis` which is set to `0` by default.

> remember, axis 0 is for rows and axis 1 is for columns.

That's why by default it checks for missing values in every row.

We can change the axis to one and we should see that `A` and `B` columns are dropped.

In [68]:
```
df.dropna(axis=1)
```

Out[68]:

	C
0	7
1	5
2	6
3	8

And `A` and `B` columns are dropped.

One last thing I want to talk about is the `threshold` argument.

> This threshold argument acts as a logical way to remove missing data.

If we set the threshold to a number then `dropna()` will only drop `rows/columns` if they have `more than or equal` that number of missing values.

For example,

In [69]:
```
df
```

Out[69]:

	A	B	C
0	1.0	2.0	7
1	2.0	NaN	5
2	NaN	NaN	6
3	3.0	4.0	8

`Row 2` has 2 missing values. So, only `row2` should be dropped if we set the threshold to `2`.

In [70]:
```
df.dropna(
    axis=0,
    thresh=2
)
```

Out[70]:

	A	B	C
0	1.0	2.0	7
1	2.0	NaN	5
3	3.0	4.0	8

I set the `threshold` to `2` which means that `dropna()` will only drop a `row` if it has `2 or more` missing values.

> Row 2 is dropped.

> Dropping rows or columns with missing can be helpful if you are working with large data sets and if you remove some rows or columns with missing data it won't affect the rest of the data.

But it's also important to keep in mind that if you are working with small/midium data sets you might want `fill` the missing values with something.

FillNa

`fillna()` is the opposite of `dropna()`.

Instead of dropping a whole row or column it will `fill` the missing values with something.

It can be anything like `0`, `mean`, `median`, `mode` etc.

Like let's say I want to fill the missing values with `"MISSING"`.

In [71]:
```
df.fillna(
    value="MISSING"
```

```
)
```

Out[71]:

	A	B	C
0	1.0	2.0	7
1	2.0	MISSING	5
2	MISSING	MISSING	6
3	3.0	4.0	8

And you can see that the `Nan` values are now replaced with `"MISSING"`.

> It also has an `inplace` argument to make the changes permanent. And like `dropna()` it also has an `axis` argument to specify which axis to fill the missing data.

In [72]:

```
df.fillna(
    value=df['A'].mean(),
    axis=0,
)
```

Out[72]:

	A	B	C
0	1.0	2.0	7
1	2.0	2.0	5
2	2.0	2.0	6
3	3.0	4.0	8

We are filling all the missing values with `mean` of column `A`.

And the good part is we can also apply this for each `columns` separately. This gives us more control of what we want to do with the missing data.

In [73]:

```
df['A'].fillna(
    value=df['A'].mean()
)
```

Out[73]:

```
0    1.0
1    2.0
2    2.0
3    3.0
Name: A, dtype: float64
```

In [74]:

```
df['B'].fillna(
    value=df['B'].mean()
)
```

Out[74]:

```
0    2.0
1    3.0
2    3.0
3    4.0
Name: B, dtype: float64
```

> We can directly apply `fillna()` or `dropna()` on specific `columns` instead of the whole `dataframe`. This will keep your data cleaner and safe from un-wanted changes.

Now, how you use these methods it's upto the datasets you are working with and one the most important things `pandas` gives you for data processing is `group by` method. Let's talk about that next.

Group By

If you have experience with `SQL` you might be familiar with `group by` method.

> Group By is a way to group data together based on a column and then perform an `aggregation(mean, sum, count, min, max)` on that group.

I think showing you would be more effective.

Let's make a `dataframe` first.

In [75]:
```
np.random.seed(101)

data = {
    'Store Name': ['Pizza Hut', 'PizzaBurg', 'Dominos', 'Pizza Hut', 'Domi
    'Location': ['Dhanmondi', 'Taltola', 'Dhanmondi', 'Mirpur', 'Mirpur',
    'Price' : np.random.randint(300, 700, size=7)
}
data
```

Out[75]:
```
{'Store Name': ['Pizza Hut',
  'PizzaBurg',
  'Dominos',
  'Pizza Hut',
  'Dominos',
  'Pizza Hut',
  'PizzaRella'],
 'Location': ['Dhanmondi',
  'Taltola',
  'Dhanmondi',
  'Mirpur',
  'Mirpur',
  'Gulshan',
  'taltola'],
 'Price': array([651, 311, 637, 626, 363, 387, 375])}
```

In [76]:
```
df = pd.DataFrame(data)
df
```

Out[76]:

	Store Name	Location	Price
0	Pizza Hut	Dhanmondi	651
1	PizzaBurg	Taltola	311
2	Dominos	Dhanmondi	637
3	Pizza Hut	Mirpur	626
4	Dominos	Mirpur	363
5	Pizza Hut	Gulshan	387
6	PizzaRella	taltola	375

Looks good.

Now we cna group the `dataframe` by the `Store Name` or `location` column.

So, let's say I want to group the `dataframe` by the `Store Name` column and see which `outlet` has the highest price.

In [77]:
```
df.groupby(['Store Name']).max(numeric_only=True)
```

Out[77]:

Store Name	Price
Dominos	637
Pizza Hut	651
PizzaBurg	311
PizzaRella	375

And you can see `dominos` has the highest.

> Now, breaking it down can be more readable. As, the locations are also grouped together with the `Store Name` column they can cause some issues because they are not numeric values. That's why I used `numeric_only=True`. This will tell `pandas` to only consider numeric values when aggregating the `dataframe`.

In [78]:
```
store_group = df.groupby('Store Name')
store_group
```

Out[78]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7047b1dd2d10>
```

`df.groupby('Store Name')` will give us a `groupby` object and this object has a lot of built in `aggregate` methods.

And `max()` is one of them. And we can use it on the `groupby` object.

In [79]:
```
store_group.max(numeric_only=True)
```

Out[79]:

	Price
Store Name	
Dominos	637
Pizza Hut	651
PizzaBurg	311
PizzaRella	375

We can find mean, sum, count, min, max, etc. on a `groupby` object.

In [80]:
```
store_group.min(numeric_only=True)
```

Out[80]:

	Price
Store Name	
Dominos	363
Pizza Hut	387
PizzaBurg	311
PizzaRella	375

In [81]:
```
store_group.mean()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/groupby.py:1944, in GroupBy._agg_py_fallback(self, how, values, ndim, a
lt)
   1943 try:
-> 1944     res_values = self._grouper.agg_series(ser, alt, preserve_dtype=
True)
   1945 except Exception as err:

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/ops.py:873, in BaseGrouper.agg_series(self, obj, func, preserve_dtype)
    871     preserve_dtype = True
--> 873 result = self._aggregate_series_pure_python(obj, func)
    875 npvalues = lib.maybe_convert_objects(result, try_float=False)

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/ops.py:894, in BaseGrouper._aggregate_series_pure_python(self, obj, fun
c)
    893 for i, group in enumerate(splitter):
--> 894     res = func(group)
    895     res = extract_result(res)

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/groupby.py:2461, in GroupBy.mean.<locals>.<lambda>(x)
   2458 else:
   2459     result = self._cython_agg_general(
   2460         "mean",
-> 2461         alt=lambda x: Series(x, copy=False).mean(numeric_only=numer
ic_only),
   2462         numeric_only=numeric_only,
   2463     )
   2464     return result.__finalize__(self.obj, method="groupby")

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/seri
es.py:6570, in Series.mean(self, axis, skipna, numeric_only, **kwargs)
   6562 @doc(make_doc("mean", ndim=1))
   6563 def mean(
   6564     self,
   (...)   6568     **kwargs,
   6569 ):
-> 6570     return NDFrame.mean(self, axis, skipna, numeric_only, **kwargs)

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/gene
ric.py:12485, in NDFrame.mean(self, axis, skipna, numeric_only, **kwargs)
  12478 def mean(
  12479     self,
  12480     axis: Axis | None = 0,
   (...)  12483     **kwargs,
  12484 ) -> Series | float:
> 12485     return self._stat_function(
  12486         "mean", nanops.nanmean, axis, skipna, numeric_only, **kwarg
s
  12487     )

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/gene
ric.py:12442, in NDFrame._stat_function(self, name, func, axis, skipna, num
eric_only, **kwargs)
  12440 validate_bool_kwarg(skipna, "skipna", none_allowed=False)
> 12442 return self._reduce(
```

```
  12443     func, name=name, axis=axis, skipna=skipna, numeric_only=numeric
_only
  12444 )

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/seri
es.py:6478, in Series._reduce(self, op, name, axis, skipna, numeric_only, f
ilter_type, **kwds)
   6474     raise TypeError(
   6475         f"Series.{name} does not allow {kwd_name}={numeric_only} "
   6476         "with non-numeric dtypes."
   6477     )
-> 6478 return op(delegate, skipna=skipna, **kwds)

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/nano
ps.py:147, in bottleneck_switch.__call__.<locals>.f(values, axis, skipna, *
*kwds)
    146 else:
--> 147     result = alt(values, axis=axis, skipna=skipna, **kwds)
    149 return result

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/nano
ps.py:404, in _datetimelike_compat.<locals>.new_func(values, axis, skipna,
mask, **kwargs)
    402     mask = isna(values)
--> 404 result = func(values, axis=axis, skipna=skipna, mask=mask, **kwarg
s)
    406 if datetimelike:

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/nano
ps.py:720, in nanmean(values, axis, skipna, mask)
    719 the_sum = values.sum(axis, dtype=dtype_sum)
--> 720 the_sum = _ensure_numeric(the_sum)
    722 if axis is not None and getattr(the_sum, "ndim", False):

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/nano
ps.py:1701, in _ensure_numeric(x)
   1699 if isinstance(x, str):
   1700     # GH#44008, GH#36703 avoid casting e.g. strings to numeric
-> 1701     raise TypeError(f"Could not convert string '{x}' to numeric")
   1702 try:

TypeError: Could not convert string 'DhanmondiMirpur' to numeric

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

TypeError                                 Traceback (most recent call last)
Cell In[81], line 1
----> 1 store_group.mean()

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/groupby.py:2459, in GroupBy.mean(self, numeric_only, engine, engine_kwa
rgs)
   2452     return self._numba_agg_general(
   2453         grouped_mean,
   2454         executor.float_dtype_mapping,
   2455         engine_kwargs,
   2456         min_periods=0,
   2457     )
   2458 else:
-> 2459     result = self._cython_agg_general(
   2460         "mean",
```

```
   2461         alt=lambda x: Series(x, copy=False).mean(numeric_only=numer
ic_only),
   2462         numeric_only=numeric_only,
   2463     )
   2464     return result.__finalize__(self.obj, method="groupby")

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/groupby.py:2005, in GroupBy._cython_agg_general(self, how, alt, numeric
_only, min_count, **kwargs)
   2002     result = self._agg_py_fallback(how, values, ndim=data.ndim, alt
=alt)
   2003     return result
-> 2005 new_mgr = data.grouped_reduce(array_func)
   2006 res = self._wrap_agged_manager(new_mgr)
   2007 if how in ["idxmin", "idxmax"]:

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/inte
rnals/managers.py:1488, in BlockManager.grouped_reduce(self, func)
   1484 if blk.is_object:
   1485     # split on object-dtype blocks bc some columns may raise
   1486     #  while others do not.
   1487     for sb in blk._split():
-> 1488         applied = sb.apply(func)
   1489         result_blocks = extend_blocks(applied, result_blocks)
   1490 else:

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/inte
rnals/blocks.py:395, in Block.apply(self, func, **kwargs)
    389 @final
    390 def apply(self, func, **kwargs) -> list[Block]:
    391     """
    392     apply the function to my values; return a block if we are not
    393     one
    394     """
--> 395     result = func(self.values, **kwargs)
    397     result = maybe_coerce_values(result)
    398     return self._split_op_result(result)

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/groupby.py:2002, in GroupBy._cython_agg_general.<locals>.array_func(val
ues)
   1999     return result
   2001 assert alt is not None
-> 2002 result = self._agg_py_fallback(how, values, ndim=data.ndim, alt=al
t)
   2003 return result

File ~/miniconda3/envs/ml_env/lib/python3.11/site-packages/pandas/core/grou
pby/groupby.py:1948, in GroupBy._agg_py_fallback(self, how, values, ndim, a
lt)
   1946     msg = f"agg function failed [how->{how},dtype->{ser.dtype}]"
   1947     # preserve the kind of exception that raised
-> 1948     raise type(err)(msg) from err
   1950 dtype = ser.dtype
   1951 if dtype == object:

TypeError: agg function failed [how->mean,dtype->object]
```

And you can see that it's giving us an error. Because mean method cannot be applied for `non-numeric` values.

That's why you should remember to use `numeric_only=True` when you group by a column.

In [82]:
```
store_group.mean(numeric_only=True)
```

Out[82]:

	Price
Store Name	
Dominos	500.000000
Pizza Hut	554.666667
PizzaBurg	311.000000
PizzaRella	375.000000

As after using a `aggregation` mathod the groupby object returns a `dataframe` we can use all the dataframe methods on it to.

In [83]:
```
agg_df = store_group.sum(numeric_only=True)
agg_df.loc['Dominos']
```

Out[83]:
```
Price    1000
Name: Dominos, dtype: int64
```

And also one very use full mathod of `dataframe` that I forgot to talk about is `describe()`.

This mathod can give use a summary of the `dataframe` like `count`, `mean`, `std`, `min`, `25%`, `50%`, `75%`, `max` etc.

For example:

In [84]:
```
df.describe()
```

Out[84]:

	Price
count	7.000000
mean	478.571429
std	151.170827
min	311.000000
25%	369.000000
50%	387.000000
75%	631.500000
max	651.000000

`Group by` also has this `method`

In [85]:
```
store_group.describe()
```

Out[85]:

	Price							
	count	mean	std	min	25%	50%	75%	max
Store Name								
Dominos	2.0	500.000000	193.747258	363.0	431.5	500.0	568.5	637.0
Pizza Hut	3.0	554.666667	145.740637	387.0	506.5	626.0	638.5	651.0
PizzaBurg	1.0	311.000000	NaN	311.0	311.0	311.0	311.0	311.0
PizzaRella	1.0	375.000000	NaN	375.0	375.0	375.0	375.0	375.0

And we can see `count`, `mean`, `std`, `min`, `25%`, `50%`, `75%`, `max` of each group.

It's a very handy method to use and you will use it a lot in your data analysis journey.

Joining & Merging

Joining and merging is another useful mathod of `pandas` that you might need for joining or merging multiple `Data-frames`.

If you have experience with `SQL` you know the basics of `join` and `merge` mathods.

But if you don't then try to practice and understand the outputs thoroughly.

I'll make three `dataframes` first.

In [86]:

```
np.random.seed(101)
df1 = pd.DataFrame({
    'A': np.random.randint(1,10,size=5),
    'B': np.random.randint(1,10,size=5),
    'C': np.random.randint(1,10,size=5),
    'D': np.random.randint(1,10,size=5)
})
df1
```

Out[86]:

	A	B	C	D
0	2	9	2	3
1	7	6	4	9
2	8	1	9	4
3	9	6	4	8
4	5	9	4	1

In [87]:

```
np.random.seed(102)
df2 = pd.DataFrame({
    'A': np.random.randint(1,10,size=5),
    'B': np.random.randint(1,10,size=5),
    'C': np.random.randint(1,10,size=5),
    'D': np.random.randint(1,10,size=5)
})
df2
```

Out[87]:

	A	B	C	D
0	1	9	1	1
1	4	9	7	1
2	3	8	3	1
3	3	5	8	7
4	3	8	4	8

In [88]:

```
np.random.seed(103)
df3 =pd.DataFrame({
    'A': np.random.randint(1,10,size=5),
    'B': np.random.randint(1,10,size=5),
    'C': np.random.randint(1,10,size=5),
    'D': np.random.randint(1,10,size=5)
})
df3
```

Out[88]:

	A	B	C	D
0	8	8	5	1
1	4	8	9	2
2	7	5	2	7
3	2	7	5	1
4	6	2	6	4

Concat

To connect multiple `dataframes` together you can use the `concat` mathod. This method will connect the `dataframes` together `by rows` and the `index` of the `dataframes` will be preserved.

In [89]:

```
pd.concat([df1,df2,df3])
```

Out[89]:

	A	B	C	D
0	2	9	2	3
1	7	6	4	9
2	8	1	9	4
3	9	6	4	8
4	5	9	4	1
0	1	9	1	1
1	4	9	7	1
2	3	8	3	1
3	3	5	8	7
4	3	8	4	8
0	8	8	5	1
1	4	8	9	2
2	7	5	2	7
3	2	7	5	1
4	6	2	6	4

As you can see, the dataframes are connected together `by rows` and the `index` of the `dataframes` are repeating. This might cause some issues later on. So, what we can do is ignore the `index` and connect the `dataframes` together `by columns`.

In [90]:

```
pd.concat([df1,df2,df3], ignore_index=True)
```

Out[90]:

	A	B	C	D
0	2	9	2	3
1	7	6	4	9
2	8	1	9	4
3	9	6	4	8
4	5	9	4	1
5	1	9	1	1
6	4	9	7	1
7	3	8	3	1
8	3	5	8	7
9	3	8	4	8
10	8	8	5	1
11	4	8	9	2
12	7	5	2	7
13	2	7	5	1
14	6	2	6	4

> The concat method has a `parameter` called `ignore_index` that you can use to ignore the index of the `dataframes` and connect them together `by columns`.

Now, what about concatting the `dataframes` `by columns`?

We can just change the `axis` parameter to `1`.

In [91]:
```
pd.concat([df1,df2,df3],axis=1)
```

Out[91]:

	A	B	C	D	A	B	C	D	A	B	C	D
0	2	9	2	3	1	9	1	1	8	8	5	1
1	7	6	4	9	4	9	7	1	4	8	9	2
2	8	1	9	4	3	8	3	1	7	5	2	7
3	9	6	4	8	3	5	8	7	2	7	5	1
4	5	9	4	1	3	8	4	8	6	2	6	4

Cancat is farely easy to unsderstand. What's hard to get your head around is `merging/joining`.

Merge

I'll simply make two `dataframes` for you.

In [92]:
```
np.random.seed(101)

left = pd.DataFrame({
    'key': [1, 2, 3, 4],
    'A': np.random.randint(1,10,size=4),
    'B': np.random.randint(1,10,size=4)
})
right = pd.DataFrame({
    'key': [1, 2, 3, 4],
    'C': np.random.randint(1,10,size=4),
    'D': np.random.randint(1,10,size=4)
})
```

In [93]:
```
left
```

Out[93]:

	key	A	B
0	1	2	5
1	2	7	9
2	3	8	6
3	4	9	1

In [94]:
```
right
```

Out[94]:

	key	C	D
0	1	6	9
1	2	9	4
2	3	2	4
3	4	4	3

Now, let's merge these two `dataframes` together.

> Merging usually happens when you want to join two `dataframes` together based on `one or more` `KEY` column(s).

If you take a look at the `dataframes` you'll see that they have a `key` column. So, we can merge these two `dataframes` together based on the `key` column.

In [95]:
```
pd.merge(left,right,how='inner',on='key')
```

Out[95]:

	key	A	B	C	D
0	1	2	5	6	9
1	2	7	9	9	4
2	3	8	6	2	4
3	4	9	1	4	3

> We have to use pd.merge() to merge `dataframes`.

Now you might ask what are these attributes?

`how` is asking you what type of join you want to do.

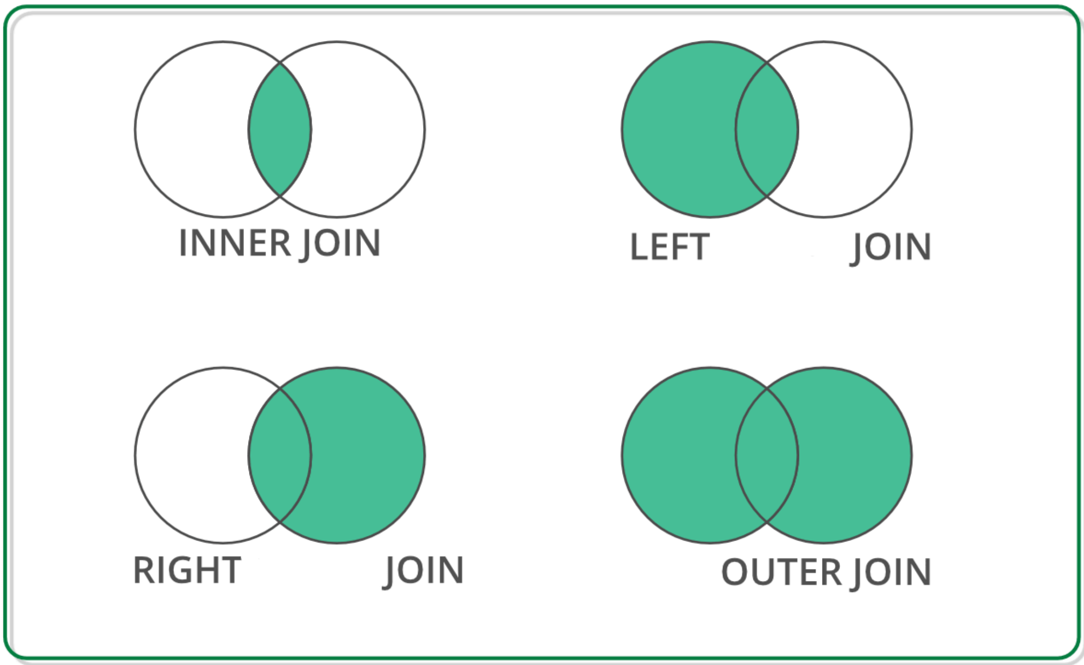
> Yes merging and joining are the 99% of the time the same thing.

`on` is asking you what column you want to join on.

And I think I should talk about the types of joins.

There are many types of joins in pandas. Most significant ones are:

1. Inner join
2. Outer join
3. Left join
4. Right join



- `Inner join` is the default join type. And it is the most common join type. When you are inner joining a `dataframe` with another `dataframe`, only the common rows between the two `dataframes` are kept.
- `Outer join` is the opposite of inner join. When you are outer joining a `dataframe` with another `dataframe`, all the rows from the two `dataframes` are kept, even if they are not common.
- `Left join` keeps all the rows from the left `dataframe` and only the common rows from the right `dataframe`.
- `Right join` keeps all the rows from the right `dataframe` and only the common rows from the left `dataframe`.

Let's see some examples to grasp the idea.

In [104…

```
np.random.seed(101)

left = pd.DataFrame({
    'key1': ['K0', 'K1', 'K2', 'K3'],
    'A': np.random.randint(1,10,size=4),
    'B': np.random.randint(1,10,size=4)
})

right = pd.DataFrame({
    'key1': ['K0', 'K2', 'K4', 'K5'],
    'C': np.random.randint(1,10,size=4),
    'D': np.random.randint(1,10,size=4)
})
```

In [111…

```
pd.merge(left, right, on='key1')
```

Out[111…

	key1	A	B	C	D
0	K0	2	5	6	9
1	K2	8	6	9	4

For multiple `keys` we can pass a `list` of `keys` to the `on` parameter.

In [112…
```
pd.merge(left, right, how='outer', on=['key1'])
```

Out[112…

	key1	A	B	C	D
0	K0	2.0	5.0	6.0	9.0
1	K1	7.0	9.0	NaN	NaN
2	K2	8.0	6.0	9.0	4.0
3	K3	9.0	1.0	NaN	NaN
4	K4	NaN	NaN	2.0	4.0
5	K5	NaN	NaN	4.0	3.0

In [113…
```
pd.merge(left, right, how='right', on=['key1'])
```

Out[113…

	key1	A	B	C	D
0	K0	2.0	5.0	6	9
1	K2	8.0	6.0	9	4
2	K4	NaN	NaN	2	4
3	K5	NaN	NaN	4	3

In [114…
```
pd.merge(left, right, how='left', on=['key1'])
```

Out[114…

	key1	A	B	C	D
0	K0	2	5	6.0	9.0
1	K1	7	9	NaN	NaN
2	K2	8	6	9.0	4.0
3	K3	9	1	NaN	NaN

Joining

Joining is exactly the same as merging except that it is a `dataframe` specific method and it's applied on `index` instead of `key`

In [115…
```
left = pd.DataFrame({'A': np.random.randint(1,10,size=5),
                     'B': np.random.randint(1,10,size=5),
                     },index=[0,2,3,4,6])

right = pd.DataFrame({'C': np.random.randint(1,10,size=4),
                      'D': np.random.randint(1,10,size=4),
                      }, index=[0,1,2,3])
```

In [118…
```
left
```

Out[118...

	A	B
0	9	9
2	4	5
3	8	4
4	1	4
6	8	8

In [119...

```
right
```

Out[119...

	C	D
0	5	5
1	9	3
2	8	8
3	7	8

In [120...

```
left.join(right)
```

Out[120...

	A	B	C	D
0	9	9	5.0	5.0
2	4	5	8.0	8.0
3	8	4	7.0	8.0
4	1	4	NaN	NaN
6	8	8	NaN	NaN

In [121...

```
left.join(right, how='outer')
```

Out[121...

	A	B	C	D
0	9.0	9.0	5.0	5.0
1	NaN	NaN	9.0	3.0
2	4.0	5.0	8.0	8.0
3	8.0	4.0	7.0	8.0
4	1.0	4.0	NaN	NaN
6	8.0	8.0	NaN	NaN

We just have to specify which data frame we want to join and what type of join we want to do.

Pandas is very important and useful library and I've said it couple of times and one of the reason behind it is the handy `methods` that it has for different operations and use cases.

Operations

Let me just show you a few examples of different operations that you can do on `dataframes`.

In [123…

```
np.random.seed(101)

import pandas as pd
df = pd.DataFrame({
    'col1': np.random.randint(1,10,size=10),
    'col2': np.random.randint(100,500,size=10),
    'col3': np.random.choice(['steve','bob','mike', 'sara'],size=10)
})

df
```

Out[123…

	col1	col2	col3
0	2	149	sara
1	7	439	mike
2	8	211	mike
3	9	236	steve
4	5	471	sara
5	9	471	sara
6	6	159	sara
7	1	144	steve
8	6	428	bob
9	9	349	sara

Head

Sometimes the dataframe is really long and you just want to see the first few rows.

You can use a method called `head()`

In [124…

```
df.head()
```

Out[124…

	col1	col2	col3
0	2	149	sara
1	7	439	mike
2	8	211	mike
3	9	236	steve
4	5	471	sara

> By default it returns the first 5 rows. You can also specify the number of rows you want to see inside the `head()` method.

Unique

Let's say you want to know if a column is categorical or not. Or you just want to see the unique values of a column.

In [126…

```
df['col3'].unique()
```

Out[126…

```
array(['sara', 'mike', 'steve', 'bob'], dtype=object)
```

If you just want to know the number of unique values in a column you use the `nunique()` method.

In [128…

```
df['col1'].nunique()
```

Out[128…

```
7
```

And if you want to know the frequency of unique values in a column you use the `value_counts()` method.

In [130…

```
df['col3'].value_counts()
```

Out[130…

```
col3
sara     5
mike     2
steve    2
bob      1
Name: count, dtype: int64
```

Applying Functions

Along side many many built in methods `pandas` has one of the most important methods called `apply()`. This method let's you apply custom functions to columns of a `dataframe`.

Let's say we have a python function that takes in a number and returns the `square` of that number.

In [132…

```
def times2(x):
    return x**2
```

Now we can pass this function to the `apply()` method to apply it to a column of a `dataframe`.

And every value of that column will be passed to the function and the function will return the `square` of that value.

In [133…

```
df['col1'].apply(times2)
```

Out[133…
```
0     4
1    49
2    64
3    81
4    25
5    81
6    36
7     1
8    36
9    81
Name: col1, dtype: int64
```

We can also pass standard python functions to the `apply()` method.

In [135…
```
df['col3'].apply(len)
```

Out[135…
```
0    4
1    4
2    4
3    5
4    4
5    4
6    4
7    5
8    3
9    4
Name: col3, dtype: int64
```

For additional information about a `dataframe` there's methods like `shape`, `columns`, `index`, `dtypes` etc.

In [136…
```
df.columns
```

Out[136…
```
Index(['col1', 'col2', 'col3'], dtype='object')
```

In [137…
```
df.index
```

Out[137…
```
RangeIndex(start=0, stop=10, step=1)
```

And as we have a linear structure of data we must have a sorting method too right?

In [138…
```
df.sort_values(by='col2') #inplace=False by default
```

Out[138...

	col1	col2	col3
7	1	144	steve
0	2	149	sara
6	6	159	sara
2	8	211	mike
3	9	236	steve
9	9	349	sara
8	6	428	bob
1	7	439	mike
4	5	471	sara
5	9	471	sara

dataframes also have a method named `isnull` that return a boolean dataframe for indicating `null` values.

In [139...

```
df.isnull()
```

Out[139...

	col1	col2	col3
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False
4	False	False	False
5	False	False	False
6	False	False	False
7	False	False	False
8	False	False	False
9	False	False	False

And with this we are at the end of this article and I'll end this article by talking about `taking inputs` or `loading data`.

Loading data

Pandas has the functionality to read data from almost every source like `csv`, `json`, `html`, `sql`, `excel` etc.

it has mathods like:

- `read_csv` for `csv` files. We will work with csv files a lot in the up coming articles.
- `read_json` for `json` files.

- `read_html` to read and load data straight from `urls` .
- `read_sql` for `sql` databases.
- `read_excel` for `excel` files. And many More. I'll show you some examples.

I made a csv file called `test.csv` (the dataframe I made above) and I'll read it using the `read_csv` method.

In [142…
```
csv_df = pd.read_csv('test.csv')
```

In [143…
```
csv_df
```

Out[143…

	Unnamed: 0	col1	col2	col3
0	0	2	149	sara
1	1	7	439	mike
2	2	8	211	mike
3	3	9	236	steve
4	4	5	471	sara
5	5	9	471	sara
6	6	6	159	sara
7	7	1	144	steve
8	8	6	428	bob
9	9	9	349	sara

And vuila! We have a dataframe. And if you take a look there is a column named `unnamed` this is index of the csv file. Pandas read it as a column.

We can drop it before loading.

In [144…
```
csv_df = pd.read_csv('test.csv', index_col=0)
```

In [145…
```
csv_df
```

Out[145…

	col1	col2	col3
0	2	149	sara
1	7	439	mike
2	8	211	mike
3	9	236	steve
4	5	471	sara
5	9	471	sara
6	6	159	sara
7	1	144	steve
8	6	428	bob
9	9	349	sara

> Can specify which column is the index by passing the `index_col` argument to the `read_csv()` method.

Now, let's say we have a `sql` database and we want to load the data from it.

It depends on the database you are using. Let's we are using `sqlite` database.

So, we just have to pass the path to the database to the `read_sql()` method.

> SQLite is very simple server database and it doen't need to be installed on your computer.

> For other databases though you need to install the driver for that database and also make a connection to the database engine.

In [149…
```
import sqlite3

conn = sqlite3.connect('test.db')
```

Here I made a simple `test.db` database and using the python built in `sqlite3` module I made a connection to the database.

In [150…
```
sql_df = pd.read_sql('SELECT * FROM test', conn)
```

We just have to pass the connection and the `sql` statement to the `read_sql()` method and `read_sql()` will return a `dataframe`.

In [152…
```
sql_df
```

Out[152…

	index	col1	col2	col3
0	0	2	149	sara
1	1	7	439	mike
2	2	8	211	mike
3	3	9	236	steve
4	4	5	471	sara
5	5	9	471	sara
6	6	6	159	sara
7	7	1	144	steve
8	8	6	428	bob
9	9	9	349	sara

There are bunch of these methods. You can find them in the [Pandas input/output documentation](#).

There very easy to use and very powerful.

And bonus information for you all. You can convert `dataframes` to `sql` or `json` or `csv` files using some methods to. I think it'll be good research for you guys, so try to find it out yourself and convert a `dataframe` to any of these formats.

And that is it!

It took longer than I thought and also the article is bigger than I expected.

Last Words

`Numpy` and `Pandas` are the most important libraries for you as you venture into the world of `data science` and `machine learning`.

It is your own responsibility to learn them, practice them and master them.

In my 4 years of programming journey I have learned that if you take a shortcut and skip something some place somewhere you'll have to come back to it later. `Numpy` and `Pandas` are no exception.

Thank you for reading this article and I hope you learned something new.

Happy Coding!