😊 GroupBy Operations

What is GroupBy Operation in Pandas?

The **GroupBy** operation in Pandas is a powerful and flexible tool used for **splitting** a dataset into groups, **applying** a function to each group independently, and **combining** the results into a final output. It is based on a concept borrowed from SQL and relational databases, often used in data aggregation and transformation tasks.

The GroupBy Process: Split–Apply–Combine

1. **Split**: The data is divided into groups based on one or more keys (column values).
2. **Apply**: A function is applied independently to each group. This function could be an aggregation (like sum or mean), a transformation, or a filtration.
3. **Combine**: The results of the function application are combined into a new DataFrame or Series.

Common Use Cases of GroupBy

- Summarizing data (e.g., average revenue by region)
- Analyzing subsets (e.g., customer behavior by segment)
- Transforming groups (e.g., normalizing scores within each category)
- Filtering groups (e.g., retaining only those groups that meet a condition)

Aggregation Functions with GroupBy

GroupBy is most commonly used with aggregation functions such as:

- sum() – computes the total for each group
- mean() – computes the average
- count() – counts non-null values
- min() / max() – finds the minimum or maximum values
- median() – computes the median
- std() / var() – computes standard deviation or variance

You can also define custom aggregation functions using user-defined functions.

Grouping Keys

The groupby() method can group by:

- A single column
- Multiple columns
- Index levels (in multi-index DataFrames)

Types of Functions You Can Apply

- **Aggregation**: Collapses the data to a single value per group.
- **Transformation**: Returns a modified version of each group with the same shape as the original.
- **Filtering**: Returns only those groups that meet a condition.
- **Iteration**: Allows looping over groups as (name, group) pairs.

Output Format

- The structure of the output depends on how you group and what operation you perform.
    - Grouping by a single column returns a Series or DataFrame with grouped index.
    - Grouping by multiple columns results in a hierarchical (multi-level) index.

Example Scenarios

- **Retail data**: Group by product category to find the total sales.
- **Employee data**: Group by department and compute average salary.
- **Student data**: Group by class and calculate the highest score.

Summary

GroupBy in Pandas allows data to be analyzed in a structured and organized way by segmenting data, applying custom or standard functions, and recombining the results. It is a fundamental tool for data summarization, transformation, and analysis in real-world datasets.

😄 To begin with the Lab

1. We started by importing NumPy and Pandas in our Jupyter notebook.

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

2. Then we created a DataFrame for our CSV file and represented in the notebook.

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

```
[3]: df
```

[3]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage

3. The code uses groupby() to group the DataFrame df by the 'model_year' column.
4. This creates a groupby object, which organizes the data by unique values in 'model_year'.
5. When .mean() is applied, it calculates the average of all numeric columns for each model year group.
6. After grouping, 'model_year' becomes the index of the resulting DataFrame, not just a column.
7. The groupby() method is useful for performing aggregate operations like sum, mean, or count on subsets of data based on a common key.

```
[4]: # Creates a groupby object waiting for an aggregate method
     df.groupby('model_year')
```

```
[4]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000246790FEC88>
```

```
[5]: # model_year becomes the index! It is NOT a column name,it is now the name of the index
     df.groupby('model_year').mean()
```

[5]:

model_year	mpg	cylinders	displacement	weight	acceleration	origin
70	17.689655	6.758621	281.413793	3372.793103	12.948276	1.310345
71	21.250000	5.571429	209.750000	2995.428571	15.142857	1.428571
72	18.714286	5.821429	218.375000	3237.714286	15.125000	1.535714
73	17.100000	6.375000	256.875000	3419.025000	14.312500	1.375000
74	22.703704	5.259259	171.740741	2877.925926	16.203704	1.666667
75	20.266667	5.600000	205.533333	3176.800000	16.050000	1.466667
76	21.573529	5.647059	197.794118	3078.735294	15.941176	1.470588
77	23.375000	5.464286	191.392857	2997.357143	15.435714	1.571429

8. The code groups the DataFrame `df` by `'model_year'` using `groupby()` and calculates the mean for each group, storing it in `avg_year`.

9. Accessing `avg_year.index` shows the unique model years, while `avg_year.columns` lists the averaged numeric columns.
10. `avg_year['mpg']` or `df.groupby('model_year').mean()['mpg']` retrieves the average miles per gallon per year.
11. `df.groupby('model_year').describe()` gives detailed statistics (count, mean, std, etc.) for each group, and `.transpose()` flips rows and columns for easier reading.
12. The `describe()` method summarizes data, and `groupby()` allows aggregation per category.

```
[6]: avg_year = df.groupby('model_year').mean()
```

```
[7]: avg_year.index
```

```
[7]: Int64Index([70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82], dtype='int64', name='model_year')
```

```
[8]: avg_year.columns
```

```
[8]: Index(['mpg', 'cylinders', 'displacement', 'weight', 'acceleration', 'origin'], dtype='object')
```

```
[9]: avg_year['mpg']
```

```
[9]: model_year
     70    17.689655
     71    21.250000
     72    18.714286
     73    17.100000
     74    22.703704
     75    20.266667
     76    21.573529
     77    23.375000
     78    24.061111
     79    25.093103
     80    33.696552
     81    30.334483
     82    31.709677
     Name: mpg, dtype: float64
```

```
[10]: df.groupby('model_year').mean()['mpg']
```

```
[10]: model_year
      70    17.689655
      71    21.250000
      72    18.714286
      73    17.100000
      74    22.703704
      75    20.266667
      76    21.573529
      77    23.375000
      78    24.061111
      79    25.093103
      80    33.696552
      81    30.334483
      82    31.709677
      Name: mpg, dtype: float64
```

```
[11]: df.groupby('model_year').describe()
```

[11]:

	mpg								cylinders		...	acceleration								origin	
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std	min	25%	50%	75%	max
model_year																					
70	29.0	17.689655	5.339231	9.0	14.000	16.00	22.000	27.0	29.0	6.758621	...	15.000	20.5	29.0	1.310345	0.603765	1.0	1.0	1.0	1.0	3.0
71	28.0	21.250000	6.591942	12.0	15.500	19.00	27.000	35.0	28.0	5.571429	...	16.125	20.5	28.0	1.428571	0.741798	1.0	1.0	1.0	2.0	3.0
72	28.0	18.714286	5.435529	11.0	13.750	18.50	23.000	28.0	28.0	5.821429	...	16.625	23.5	28.0	1.535714	0.792658	1.0	1.0	1.0	2.0	3.0
73	40.0	17.100000	4.700245	11.0	13.000	16.00	20.000	29.0	40.0	6.375000	...	16.000	21.0	40.0	1.375000	0.667467	1.0	1.0	1.0	2.0	3.0
74	27.0	22.703704	6.420010	13.0	16.000	24.00	27.000	32.0	27.0	5.259259	...	17.000	21.0	27.0	1.666667	0.832050	1.0	1.0	1.0	2.0	3.0
75	30.0	20.266667	4.940566	13.0	16.000	19.50	23.000	33.0	30.0	5.600000	...	17.375	21.0	30.0	1.466667	0.730297	1.0	1.0	1.0	2.0	3.0
76	34.0	21.573529	5.889297	13.0	16.750	21.00	26.375	33.0	34.0	5.647059	...	17.550	22.2	34.0	1.470588	0.706476	1.0	1.0	1.0	2.0	3.0

```
[12]: df.groupby('model_year').describe().transpose()
```

[12]:

	model_year	70	71	72	73	74	75	76	77	78	79	
mpg	count	29.000000	28.000000	28.000000	40.000000	27.000000	30.000000	34.000000	28.000000	36.000000	29.000000	29.000
	mean	17.689655	21.250000	18.714286	17.100000	22.703704	20.266667	21.573529	23.375000	24.061111	25.093103	33.696
	std	5.339231	6.591942	5.435529	4.700245	6.420010	4.940566	5.889297	6.675862	6.898044	6.794217	7.037
	min	9.000000	12.000000	11.000000	11.000000	13.000000	13.000000	13.000000	15.000000	16.200000	15.500000	19.100
	25%	14.000000	15.500000	13.750000	13.000000	16.000000	16.000000	16.750000	17.375000	19.350000	19.200000	29.800
	50%	16.000000	19.000000	18.500000	16.000000	24.000000	19.500000	21.000000	21.750000	20.700000	23.900000	32.700
	75%	22.000000	27.000000	23.000000	20.000000	27.000000	23.000000	26.375000	30.000000	28.000000	31.800000	38.100
	max	27.000000	35.000000	28.000000	29.000000	32.000000	33.000000	33.000000	36.000000	43.100000	37.300000	46.600
	count	29.000000	28.000000	28.000000	40.000000	27.000000	30.000000	34.000000	28.000000	36.000000	29.000000	29.000

13. The code groups the DataFrame df by both 'model_year' and 'cylinders' using groupby(['model_year', 'cylinders']), then calculates the mean for each group of these combinations.
14. This results in a multi-level (hierarchical) index where each unique pair of model_year and cylinders defines a group.
15. The .mean() function computes the average of all numeric columns within each group.
16. When .index is accessed, it returns a MultiIndex object representing the paired grouping keys (model year and cylinder count) used for aggregation.

```
[13]: df.groupby(['model_year','cylinders']).mean()
```

[13]:

		mpg	displacement	weight	acceleration	origin
model_year	cylinders					
70	4	25.285714	107.000000	2292.571429	16.000000	2.285714
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
71	4	27.461538	101.846154	2056.384615	16.961538	1.923077
	6	18.000000	243.375000	3171.875000	14.750000	1.000000
	8	13.428571	371.714286	4537.714286	12.214286	1.000000
72	3	19.000000	70.000000	2330.000000	13.500000	3.000000
	4	23.428571	111.535714	2382.642857	17.214286	1.928571

```
[14]: df.groupby(['model_year','cylinders']).mean().index
```

```
[14]: MultiIndex([(70, 4),
                  (70, 6),
                  (70, 8),
                  (71, 4),
                  (71, 6),
                  (71, 8),
                  (72, 3),
                  (72, 4),
                  (72, 8),
                  (73, 3),
                  (73, 4),
                  (73, 6),
                  (73, 8),
                  (74, 4),
                  (74, 6),
                  (74, 8),
                  (75, 4),
                  (75, 6),
```

17. The code groups the DataFrame `df` by both `'model_year'` and `'cylinders'`, calculates the mean of numeric columns for each group, and stores the result in `year_cyl`.
18. The resulting `year_cyl` DataFrame has a **MultiIndex**, meaning the index is made up of multiple levels—specifically, `'model_year'` and `'cylinders'`.

- `year_cyl.index` returns the full MultiIndex object.
- `year_cyl.index.levels` gives the unique values at each index level (all unique model years and cylinder counts).
- `year_cyl.index.names` shows the names of the levels in the MultiIndex, which are `'model_year'` and `'cylinders'`.

```
[15]: year_cyl = df.groupby(['model_year','cylinders']).mean()
```

```
[16]: year_cyl
```

[16]:

		mpg	displacement	weight	acceleration	origin
model_year	**cylinders**					
70	**4**	25.285714	107.000000	2292.571429	16.000000	2.285714
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
71	**4**	27.461538	101.846154	2056.384615	16.961538	1.923077
	6	18.000000	243.375000	3171.875000	14.750000	1.000000
	8	13.428571	371.714286	4537.714286	12.214286	1.000000
72	**3**	19.000000	70.000000	2330.000000	13.500000	3.000000
	4	23.428571	111.535714	2382.642857	17.214286	1.928571

```
[17]: year_cyl.index
```

```
[17]: MultiIndex([(70, 4),
                  (70, 6),
                  (70, 8),
                  (71, 4),
                  (71, 6),
                  (71, 8),
                  (72, 3),
                  (72, 4),
                  (72, 8),
                  (73, 3),
                  (73, 4),
                  (73, 6),
                  (73, 8),
                  (74, 4),
                  (74, 6),
                  (74, 8),
                  (75, 4),
                  (75, 6),
```

```
[18]: year_cyl.index.levels
```

```
[18]: FrozenList([[70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82], [3, 4, 5, 6, 8]])
```

```
[19]: year_cyl.index.names
```

```
[19]: FrozenList(['model_year', 'cylinders'])
```

19. The code displays and accesses data from year_cyl, a DataFrame grouped by 'model_year' and 'cylinders' with a MultiIndex. year_cyl.head() shows the first five rows. year_cyl.loc[70] retrieves all rows for model year 1970 across all cylinder counts.
20. year_cyl.loc[[70,72]] gets all rows for model years 1970 and 1972.
21. year_cyl.loc[(70,8)] drills down to a specific group where the model year is 1970 and cylinders are 8.
22. The .loc[] method accesses rows by label, and with MultiIndex, you can specify one or multiple levels.

```
[20]: year_cyl.head()
```

[20]:

		mpg	displacement	weight	acceleration	origin
model_year	**cylinders**					
70	**4**	25.285714	107.000000	2292.571429	16.000000	2.285714
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
71	**4**	27.461538	101.846154	2056.384615	16.961538	1.923077
	6	18.000000	243.375000	3171.875000	14.750000	1.000000

[21]:
```
year_cyl.loc[70]
```

[21]:

cylinders	mpg	displacement	weight	acceleration	origin
4	25.285714	107.000000	2292.571429	16.000000	2.285714
6	20.500000	199.000000	2710.500000	15.500000	1.000000
8	14.111111	367.555556	3940.055556	11.194444	1.000000

[22]:
```
year_cyl.loc[[70,72]]
```

[22]:

model_year	cylinders	mpg	displacement	weight	acceleration	origin
70	4	25.285714	107.000000	2292.571429	16.000000	2.285714
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
	3	19.000000	70.000000	2330.000000	13.500000	3.000000
72	4	23.428571	111.535714	2382.642857	17.214286	1.928571
	8	13.615385	344.846154	4228.384615	13.000000	1.000000

[23]:
```
year_cyl.loc[(70,8)]
```

[23]:
```
mpg              14.111111
displacement    367.555556
weight         3940.055556
acceleration     11.194444
origin            1.000000
Name: (70, 8), dtype: float64
```

23. The code uses advanced MultiIndex operations. year_cyl.xs(key=70, level='model_year') selects all rows with model year 70; xs() allows selecting data at a specific index level.
24. Similarly, xs(key=4, level='cylinders') returns average values for 4-cylinder cars across all years.
25. Filtering df for 6 or 8 cylinders, then grouping, gives targeted means.
26. swaplevel() switches the order of MultiIndex levels, making 'cylinders' come before 'model_year'.
27. Finally, sort_index(level='model_year', ascending=False) sorts the grouped data by year in descending order.
28. These tools help slice, rearrange, and organize multi-indexed DataFrames efficiently.

[24]:
```
year_cyl.xs(key=70,axis=0,level='model_year')
```

[24]:

cylinders	mpg	displacement	weight	acceleration	origin
4	25.285714	107.000000	2292.571429	16.000000	2.285714
6	20.500000	199.000000	2710.500000	15.500000	1.000000
8	14.111111	367.555556	3940.055556	11.194444	1.000000

[25]:
```
# Mean column values for 4 cylinders per year
year_cyl.xs(key=4,axis=0,level='cylinders')
```

[25]:

model_year	mpg	displacement	weight	acceleration	origin
70	25.285714	107.000000	2292.571429	16.000000	2.285714
71	27.461538	101.846154	2056.384615	16.961538	1.923077
72	23.428571	111.535714	2382.642857	17.214286	1.928571
73	22.727273	109.272727	2338.090909	17.136364	2.000000
74	27.800000	96.533333	2151.466667	16.400000	2.200000
75	25.250000	114.833333	2489.250000	15.833333	2.166667
76	26.766667	106.333333	2306.600000	16.866667	1.866667
77	29.107143	106.500000	2205.071429	16.064286	1.857143

[26]:
```
df[df['cylinders'].isin([6,8])].groupby(['model_year','cylinders']).mean()
```

[26]:

model_year	cylinders	mpg	displacement	weight	acceleration	origin
70	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
71	6	18.000000	243.375000	3171.875000	14.750000	1.000000
	8	13.428571	371.714286	4537.714286	12.214286	1.000000
72	8	13.615385	344.846154	4228.384615	13.000000	1.000000
73	6	19.000000	212.250000	2917.125000	15.687500	1.250000
	8	13.200000	365.250000	4279.050000	12.250000	1.000000
74	6	17.857143	230.428571	3320.000000	16.857143	1.000000

[27]:
```
year_cyl.swaplevel().head()
```

[27]:

cylinders	model_year	mpg	displacement	weight	acceleration	origin
4	70	25.285714	107.000000	2292.571429	16.000000	2.285714
6	70	20.500000	199.000000	2710.500000	15.500000	1.000000
8	70	14.111111	367.555556	3940.055556	11.194444	1.000000
4	71	27.461538	101.846154	2056.384615	16.961538	1.923077
6	71	18.000000	243.375000	3171.875000	14.750000	1.000000

```
[28]: year_cyl.sort_index(level='model_year',ascending=False)
```

	3	18.000000	70.000000	2124.000000	13.500000	3.000000
	8	13.615385	344.846154	4228.384615	13.000000	1.000000
72	4	23.428571	111.535714	2382.642857	17.214286	1.928571
	3	19.000000	70.000000	2330.000000	13.500000	3.000000
	8	13.428571	371.714286	4537.714286	12.214286	1.000000
71	6	18.000000	243.375000	3171.875000	14.750000	1.000000
	4	27.461538	101.846154	2056.384615	16.961538	1.923077
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
70	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	4	25.285714	107.000000	2292.571429	16.000000	2.285714

29. The code demonstrates using the agg() function for multiple aggregations.
30. year_cyl.sort_index(level='cylinders', ascending=False) sorts the MultiIndex by the 'cylinders' level in descending order.
31. df.agg(['median','mean']) computes both the median and mean for all numeric columns.
32. Selecting [['mpg','weight']] limits aggregation to those two columns.
33. Using a dictionary like {'mpg': ['median', 'mean'], 'weight': ['mean', 'std']} lets you apply different functions to specific columns.
34. The final line applies these aggregations grouped by 'model_year'. The agg() function allows flexible, column-specific summary statistics.

```
[29]: year_cyl.sort_index(level='cylinders',ascending=False)
```

[29]:

		mpg	displacement	weight	acceleration	origin
model_year	cylinders					
81	8	26.600000	350.000000	3725.000000	19.000000	1.000000
79	8	18.630000	321.400000	3862.900000	15.400000	1.000000
78	8	19.050000	300.833333	3563.333333	13.266667	1.000000
77	8	16.000000	335.750000	4177.500000	13.662500	1.000000
76	8	14.666667	324.000000	4064.666667	13.222222	1.000000
75	8	15.666667	330.500000	4108.833333	13.166667	1.000000
74	8	14.200000	315.200000	4438.400000	14.700000	1.000000
73	8	13.200000	365.250000	4279.050000	12.250000	1.000000

[33]:
```
df
```

[33]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
396	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
397	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

398 rows × 9 columns

[35]:
```
# These strings need to match up with built-in method names
df.agg(['median','mean'])
```

[35]:

	mpg	cylinders	displacement	weight	acceleration	model_year	origin
median	23.000000	4.000000	148.500000	2803.500000	15.50000	76.00000	1.000000
mean	23.514573	5.454774	193.425879	2970.424623	15.56809	76.01005	1.572864

[41]:
```
df.agg(['sum','mean'])[['mpg','weight']]
```

[41]:

	mpg	weight
sum	9358.800000	1.182229e+06
mean	23.514573	2.970425e+03

[43]:
```
df.agg({'mpg':['median','mean'],'weight':['mean','std']})
```

[43]:

	mpg	weight
mean	23.514573	2970.424623
median	23.000000	NaN
std	NaN	846.841774

[44]:
```
df.groupby('model_year').agg({'mpg':['median','mean'],'weight':['mean','std']})
```

[44]:

	mpg		weight	
	median	mean	mean	std
model_year				
70	16.00	17.689655	3372.793103	852.868663
71	19.00	21.250000	2995.428571	1061.830859
72	18.50	18.714286	3237.714286	974.520960
73	16.00	17.100000	3419.025000	974.809133
74	24.00	22.703704	2877.925926	949.308571
75	19.50	20.266667	3176.800000	765.179781
76	21.00	21.573529	3078.735294	821.371481
77	21.75	23.375000	2997.357143	912.825902
78	20.70	24.061111	2861.805556	626.023907
79	23.90	25.093103	3055.344828	747.881497
80	32.70	33.696552	2436.655172	432.235491
81	31.60	30.334483	2522.931034	533.600501
82	32.00	31.709677	2453.548387	354.276713