

Lesson 09: GroupBy Operations

Data Science with Python – BSc Course

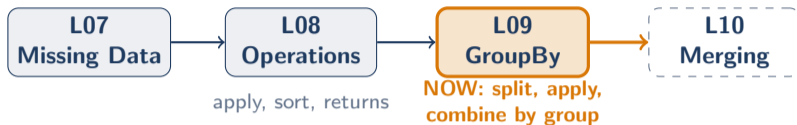
Data Science Program

BSc Course

45 Minutes

Where We Are

Last time: you transformed columns. Now: transform groups.



You can compute returns for one stock. But what if you have 500 stocks across 11 sectors? Computing the average return *per sector* requires a fundamentally different pattern: `groupby()`.

`groupby` = the SQL GROUP BY of Python. The most powerful pandas pattern.

Learning Objectives

After this lesson, you will be able to:

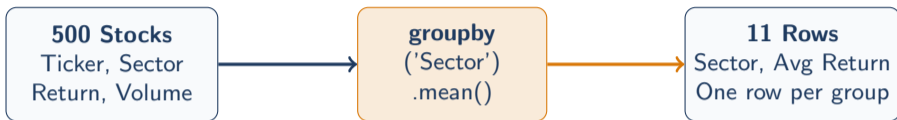
- **Explain** the split-apply-combine paradigm
- **Use** `groupby()` with aggregation functions (`mean`, `sum`, `std`)
- **Distinguish** `agg()` from `transform()` (reduce vs broadcast)
- **Apply** multi-column grouping for hierarchical analysis
- **Build** sector performance analysis with pivot tables

Finance Application: Analyze risk-return by sector, compute portfolio-level statistics, build sector heatmaps.

80% of real data analysis involves some form of `groupby`

How Do You Compute Average Return BY Sector?

The question that `groupby` was invented to answer



500 rows → 11 rows. One number per sector.

Without `groupby`, you would need a loop over each sector. With `groupby`: one line of code.

`groupby('Sector')['Return'].mean()` – the most common pandas one-liner

```
df.groupby('Sector')['Return'].mean()
```

This three-step mental model applies to SQL GROUP BY, Excel pivot tables, R's dplyr, and pandas groupby.

Split-Apply-Combine: invented by Hadley Wickham, implemented everywhere

1. SPLIT → 2. APPLY → 3. COMBINE

group keys; [step, draw=amber, fill=amber!10] (apply) at (4.5,0) **APPLY**

Run function on

each group; [step] (combine) at (9,0) **COMBINE**

Merge results

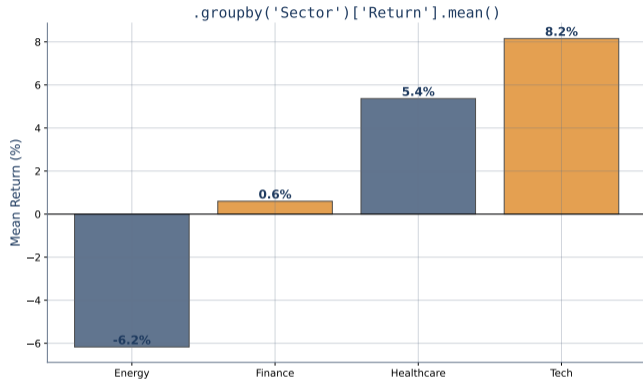
into one table; [-i, navy, very thick] (split) – (apply); [-i, amber, very thick] (apply) – (combine);

[slate, font=, align=center] at (0,-1.2) Tech: [AAPL, MSFT]

Fin: [JPM, GS]; [slate, font=, align=center] at (4.5,-1.2) Tech: mean(return)

Fin: mean(return); [slate, font=, align=center] at (9,-1.2) Tech: 0.12

Aggregation: Mean by Group



`groupby('Sector')['Return'].mean()` – one value per group

Multiple Aggregations with agg()

Apply several functions at once

```
# Multiple functions on one column
df.groupby('Sector')['Return'].agg(['mean', 'std', 'count'])

# Named aggregation (clean output)
df.groupby('Sector').agg(
    avg_return=('Return', 'mean'),
    volatility=('Return', 'std'),
    total_volume=('Volume', 'sum')
)
```

Built-in functions:

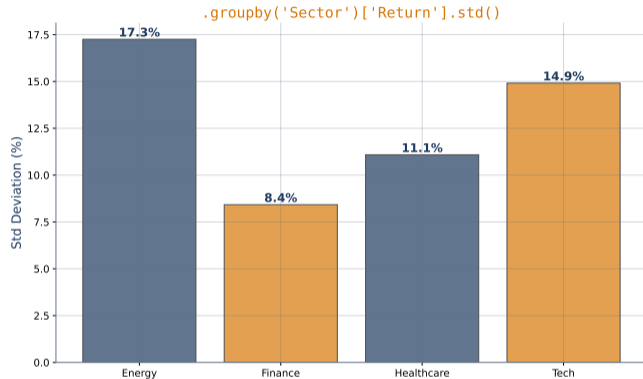
mean, sum, std, min, max,
count, first, last, median

Custom functions:

```
.agg(lambda x: x.quantile(0.95))
```

Named aggregation gives clean column names – always prefer it

Aggregation: Volatility by Group

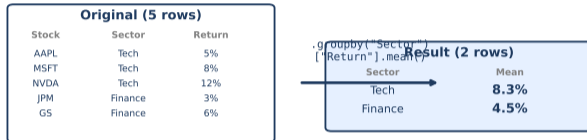


`std()` measures return dispersion within each sector

agg() vs transform(): Reduce vs Broadcast

agg() - Aggregation

Reduces each group to a single value



Use Case: Summary statistics, reports

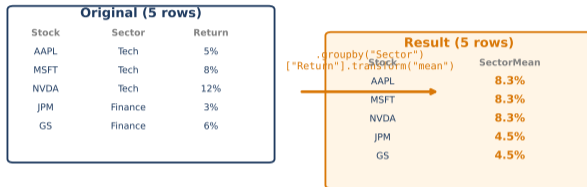
```
sector_avg = df.groupby('Sector')['Return'].mean()
```

agg() reduces each group to one value (N groups → N rows)

transform(): Keep Original Shape

transform() - Transformation

Returns same shape as input



Use Case: Adding group-level stats to each row

```
df['SectorMean'] = df.groupby('Sector')['Return'].transform('mean')
```

transform() broadcasts group result back – same length as input

Filtering Groups

Keep only groups that meet a condition

```
# Keep sectors with positive average return
df.groupby('Sector').filter(
    lambda x: x['Return'].mean() > 0)

# Keep groups with at least 10 observations
df.groupby('Sector').filter(
    lambda x: len(x) >= 10)

# Keep sectors with low volatility
df.groupby('Sector').filter(
    lambda x: x['Return'].std() < 0.02)
```

Key difference:

`filter()` returns **individual rows** (not group summaries). Only rows belonging to qualifying groups are kept.

`filter()` selects groups, not rows. All rows within qualifying groups are returned.

Checkpoint: Test Your Understanding

Q1: `agg()` vs `transform()` – which one returns the original DataFrame shape?

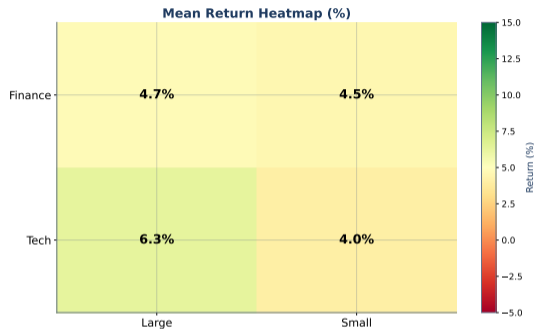
Q2: You want to z-score normalize returns within each sector. Which method: `agg`, `transform`, or `filter`?

Q3: What does `groupby('Sector').filter(lambda x: len(x) >= 5)` return?

Think for 30 seconds. Q1: `transform()`. Q2: `transform` (need original-length output). Q3: All rows from sectors with ≥ 5 stocks.

`agg` reduces, `transform` broadcasts, `filter` selects groups

Pivot Tables



`pd.pivot_table()` is groupby + reshape in one step

Pivot Tables and Crosstab

Two ways to reshape grouped data

Pivot Table:

```
pd.pivot_table(  
    df,  
    values='Return',  
    index='Sector',  
    columns='Year',  
    aggfunc='mean')
```

One function, one output table.

Crosstab:

```
pd.crosstab(  
    df['Sector'],  
    df['Signal'],  
    normalize='index')
```

Frequency table. `normalize` for proportions.

When to use which: Pivot table = aggregating numeric values. Crosstab = counting categories.

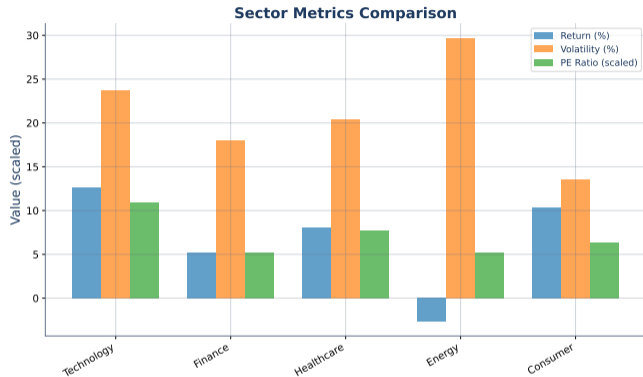
Pivot tables are the pandas equivalent of Excel pivot tables

Finance: Sector Risk-Return



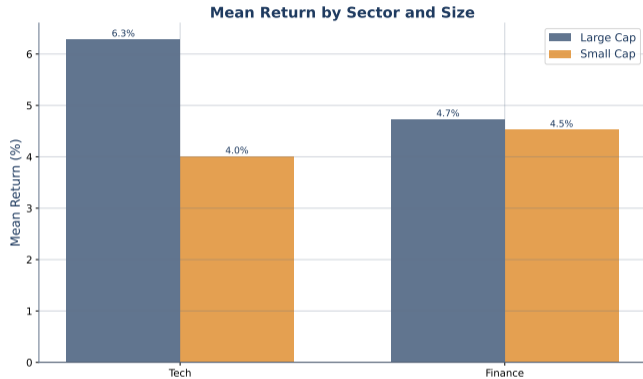
Scatter mean return vs std by sector – the classic risk-return plot

Finance: Sector Performance Metrics



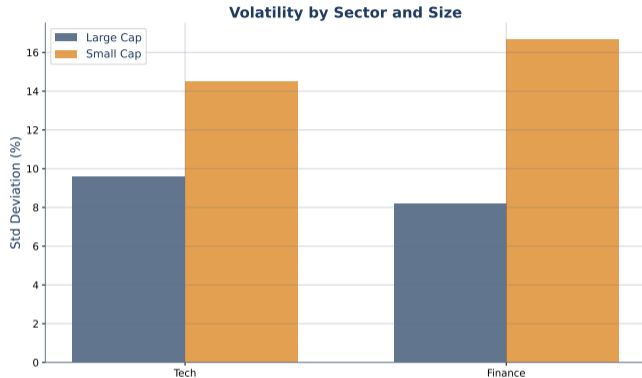
Multiple aggregations per sector using named `agg()`

Multi-Column GroupBy



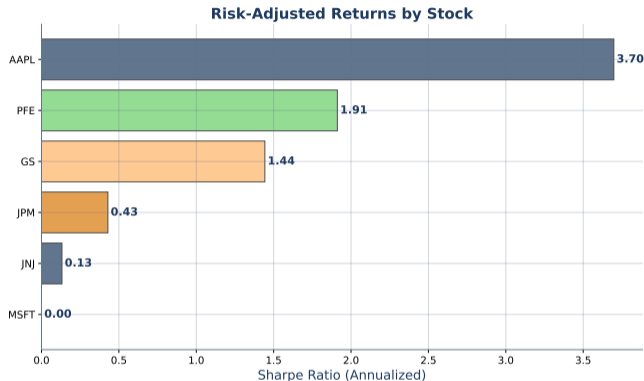
`groupby(['Sector', 'Year'])` creates hierarchical groups

Finance: Volatility by Sector and Year



Multi-group analysis reveals time-varying sector risk

Finance: Sharpe Ratio by Sector



$\text{Sharpe} = \text{mean}(\text{return}) / \text{std}(\text{return})$ – risk-adjusted performance per group

Finance: Rolling Statistics by Group

Combine groupby with rolling for per-stock analysis

```
# 20-day rolling volatility per stock
df['RollingVol'] = df.groupby('Ticker')['Return']
    .transform(lambda x: x.rolling(20).std())

# Cumulative return per stock
df['CumReturn'] = df.groupby('Ticker')['Return']
    .transform(lambda x: (1 + x).cumprod() - 1)

# Z-score within sector (de-mean by group)
df['ZReturn'] = df.groupby('Sector')['Return']
    .transform(lambda x: (x - x.mean()) / x.std())
```

Pattern: `groupby().transform()` keeps individual rows while computing group-level statistics.

transform + rolling = per-stock time series analysis in one line

GroupBy Patterns Reference

Basic:

```
df.groupby('Sector')['Return'].mean()
```

Multiple functions:

```
.agg(['mean', 'std', 'count'])
```

Named agg:

```
.agg(avg=('Return', 'mean'))
```

Performance: Built-in functions (mean, sum, std) are optimized. Custom apply() is 10–100x slower. Use built-ins when possible.

Custom function:

```
.apply(lambda x: x.quantile(0.95))
```

Filter groups:

```
.filter(lambda x: x['R'].mean() > 0.05)
```

Tips:

```
as_index=False | sort=False
```

Built-in agg functions are Cython-optimized; lambda/apply runs Python loops

Hands-on Exercise (25 min)

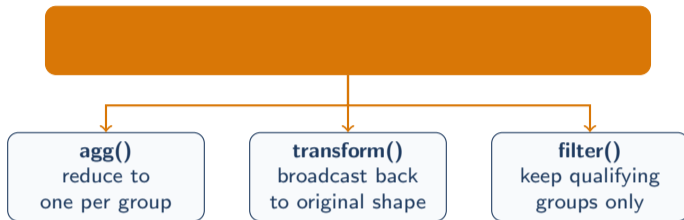
Build a sector performance analysis:

1. Load multi-stock data. Group by Sector, compute mean return.
2. Use `agg()` to compute mean, std, and count per sector.
3. Create a risk-return scatter: `x = std`, `y = mean return`, `label = sector`.
4. Use `transform()` to z-score normalize returns within sector.
5. Build a pivot table: sectors as rows, years as columns, mean return as values.
6. Use `filter()` to keep only sectors with average return > 0 .
7. Compute Sharpe ratio per sector: `mean / std * sqrt(252)`

Bonus: Add `.transform(lambda x: x.rolling(20).std())` for per-stock volatility.

groupby is the foundation of portfolio analytics and sector analysis

The Big Idea



One pattern, three flavors. Covers 80% of real-world data analysis.

GroupBy: the single most important pandas concept after DataFrames themselves

Lesson Summary

Key Takeaways:

- `groupby()` implements split-apply-combine: group → function → result
- `agg()` reduces each group to one row; `transform()` keeps original shape
- Named aggregation (`.agg(name=('col', 'func'))`) gives clean output
- `filter()` keeps all rows from groups that meet a condition
- Pivot tables reshape grouped data into row/column format

Up Next: L10 – Merging and Joining

You have one DataFrame per source. How do you combine stock prices with fundamentals from another table? `merge()` and `join()`.

L09 analyzes within one table. L10 combines multiple tables.