Lecture 20 Handout

Data Aggregation and Group Operations

Advanced Analysis Through Split-Apply-Combine

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 7, Section 7.14 - Data Analysis with pandas (GroupBy and Aggregation)
Reference Notebooks: `ch07/07_14.02.ipynb` for DataFrame operations and aggregation

Learning Objectives

By the end of this lecture, you will be able to:

1. **Perform multi-column grouping** to analyze data across multiple categorical dimensions simultaneously
2. **Apply multiple aggregations** using agg() with lists and dictionaries for comprehensive statistical summaries
3. **Create custom aggregation functions** using lambda expressions and named aggregations for specialized calculations
4. **Filter groups** using the filter() method to select groups meeting specific criteria
5. **Transform groups** using transform() to broadcast group-level calculations back to individual rows
6. **Build pivot tables** using pivot_table() to reshape data for multi-dimensional analysis
7. **Create cross-tabulations** using pd.crosstab() for frequency analysis of categorical variables
8. **Construct complete analysis workflows** combining multiple aggregation techniques for real-world data analysis

Prerequisites Review

Building on Your Complete pandas Foundation:

From Lectures 1-11, you have all Python fundamentals. From Lectures 12-13, you mastered NumPy arrays including statistical operations and boolean indexing.

From Lecture 16, you mastered pandas Series: creating Series with custom indices, accessing elements with .loc[] and .iloc[], statistical methods (count(), mean(), std(), describe()), and understanding NaN for missing data.

From Lecture 17, you mastered pandas DataFrames: creating DataFrames from dictionaries, accessing data with .loc[] and .iloc[], boolean indexing with multiple conditions, statistical methods with axis parameter, and sorting with sort_values().

From Lecture 18, you mastered data cleaning: detecting missing data with isna() and notna(), handling missing data with dropna() and fillna(), removing duplicates, and validating data types.

From Lecture 19, you learned the basics of groupby(): the split-apply-combine concept, single aggregations with mean(), sum(), count(), and size(), and basic multiple aggregations with agg(). You also learned to read and write CSV files.

Transformation Goal: Evolve from **basic single-column groupby with simple aggregations** to **sophisticated multi-dimensional analysis with pivot tables, custom functions, and group transformations** that answer complex business questions.

1 Part 1: Introduction to Advanced Aggregation and Multi-Column Grouping

1.1 Why Basic GroupBy Isn't Enough

In Lecture 19, you learned the fundamentals of groupby(): splitting data by one column, applying an aggregation like mean or sum, and combining results. This handles many common questions like "what's the average grade per major?" But real-world analysis demands more. Consider these business questions: "What's the average salary by department AND job level?" (two grouping columns). "For each product category, show me total sales, average price, and count of transactions" (multiple different aggregations). "Which stores have above-average monthly sales?" (filtering groups). "How does each employee's salary compare to their department average?" (transforming with group statistics).

These questions require advanced aggregation techniques that go beyond basic groupby. This lecture teaches you to perform multi-dimensional analysis, apply multiple different aggregations simultaneously, create custom calculations, filter and transform groups, and reshape data with pivot tables. By the end, you'll be able to answer complex analytical questions that require combining multiple techniques.

1.2 The Split-Apply-Combine Paradigm Extended

The split-apply-combine paradigm from Lecture 19 extends naturally to advanced operations. The SPLIT phase can now divide data by multiple columns, creating groups defined by combinations of categories (like Department-AND-JobLevel). The APPLY phase becomes more flexible: instead of just one aggregation, you can apply multiple functions, custom calculations, filters that keep or discard entire groups, or transformations that maintain the original data shape. The COMBINE phase assembles results that might be summaries, filtered subsets, or transformed DataFrames.

Understanding this extended paradigm helps you choose the right technique for each analytical question. Summarizing groups? Use agg(). Filtering groups by criteria? Use filter().

Adding group statistics to individual rows? Use transform(). Reshaping for cross-dimensional analysis? Use pivot_table().

1.3 Grouping by Multiple Columns

When you group by multiple columns, pandas creates groups for each unique combination of values in those columns. If you have 3 departments and 4 job levels, you could have up to 12 groups (3 x 4). This lets you analyze data across multiple categorical dimensions simultaneously.

```
import pandas as pd
import numpy as np

# Employee salary data with department and job level
employees = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve',
             'Frank', 'Grace', 'Henry'],
    'Department': ['Sales', 'Sales', 'Engineering', 'Engineering',
                   'Sales', 'Engineering', 'Sales', 'Engineering'],
    'Level': ['Senior', 'Junior', 'Senior', 'Junior',
              'Senior', 'Senior', 'Junior', 'Junior'],
    'Salary': [85000, 55000, 95000, 60000, 82000, 92000, 58000, 62000]
})

print("Employee data:")
print(employees)

# Group by both Department AND Level
dept_level = employees.groupby(['Department', 'Level'])

# Calculate mean salary for each combination
mean_salaries = dept_level['Salary'].mean()
print("\nAverage salary by Department and Level:")
print(mean_salaries)
```

The result is a Series with a MultiIndex - a hierarchical index with two levels (Department, Level). Each row represents one combination: Engineering-Junior, Engineering-Senior, Sales-Junior, Sales-Senior. This reveals that Senior employees earn more than Junior employees in both departments, and Engineering salaries are higher than Sales at both levels. You couldn't see these patterns with single-column grouping.

1.4 Understanding MultiIndex Results

Multi-column groupby produces MultiIndex results that require special access techniques. The hierarchical structure organizes data by nested categories, making it compact but requiring you to understand how to navigate it.

```
# Continue with dept_level groupby
mean_salaries = dept_level['Salary'].mean()

# Access specific combinations using tuple indexing
eng_senior = mean_salaries[('Engineering', 'Senior')]
print(f"Engineering Senior average: ${eng_senior:,.0f}")

# Convert MultiIndex to regular columns with reset_index()
# This creates a flat DataFrame that's easier to work with
salary_df = mean_salaries.reset_index()
salary_df.columns = ['Department', 'Level', 'Avg_Salary']
print("\nAs flat DataFrame:")
print(salary_df)

# Now you can filter easily
```

```
high_salaries = salary_df[salary_df['Avg_Salary'] > 70000]
print("\nCombinations with high salaries:")
print(high_salaries)
```

The `reset_index()` method converts the MultiIndex back to regular columns, creating a flat DataFrame that's easier to filter, sort, and manipulate. This is a common pattern: groupby with multiple columns, aggregate, then reset_index() to get a clean results table.

1.5 Counting Group Sizes

Understanding how many records fall into each group is essential for interpreting your aggregations. A mean based on 100 records is more reliable than one based on 2 records. Use `size()` to count records per group.

```
# Count employees in each Department-Level combination
group_sizes = employees.groupby(['Department', 'Level']).size()
print("Employees per group:")
print(group_sizes)

# Convert to DataFrame for clarity
size_df = group_sizes.reset_index(name='Count')
print("\nAs DataFrame:")
print(size_df)

# Combine count with mean to show both
summary = employees.groupby(['Department', 'Level']).agg({
    'Salary': ['mean', 'count']
})
print("\nMean and count together:")
print(summary)
```

Now you can see that Engineering-Junior has 2 employees with average $61,000, while Sales-Senior has 2 employees with average $83,500. The counts validate that your groups have reasonable sample sizes for meaningful analysis.

1.6 Part 1 Exercise

Exercise: Using the employees DataFrame from this section, perform multi-column groupby on Department and Level to find: (1) the maximum salary for each combination, (2) the total count of employees per combination. Then use reset_index() to convert the result to a flat DataFrame and filter to show only combinations with average salary above $65,000.

2 Part 2: Multiple Aggregations with agg()

2.1 Applying Multiple Functions to One Column

Often you need several statistics for the same column - not just the mean, but also the min, max, and count. The `agg()` method accepts a list of function names to apply multiple aggregations to a single column.

```
# Multiple statistics for Salary by Department
salary_stats = employees.groupby('Department')['Salary'].agg([
    'mean',
    'min',
    'max',
    'std',
    'count'
])

```

```
print("Salary statistics by Department:")
print(salary_stats)

# Round for readability
salary_stats = salary_stats.round(0)
print("\nRounded:")
print(salary_stats)
```

Each function becomes a column in the result. You can immediately see that Engineering has higher mean and max salaries, but also slightly higher standard deviation (more variability). Sales has tighter clustering around the mean. This multi-statistic view gives you a complete picture of each group's distribution.

2.2 Different Aggregations for Different Columns

Different columns often need different aggregations based on what they represent. Salary might need mean and sum, while employee count needs just count. Pass a dictionary to `agg()` mapping column names to their aggregation functions.

```
# Sales data with multiple columns needing different aggregations
sales = pd.DataFrame({
    'Region': ['East', 'East', 'West', 'West', 'East', 'West'],
    'Product': ['Widget', 'Gadget', 'Widget', 'Gadget',
                'Widget', 'Widget'],
    'Units': [100, 150, 200, 120, 180, 90],
    'Revenue': [1000, 2250, 2000, 1800, 1800, 900],
    'Profit': [200, 450, 400, 360, 360, 180]
})

print("Sales data:")
print(sales)

# Different aggregations per column
region_summary = sales.groupby('Region').agg({
    'Units': 'sum',         # Total units sold
    'Revenue': 'sum',       # Total revenue
    'Profit': ['sum', 'mean']  # Total and average profit
})

print("\nRegion summary:")
print(region_summary)
```

This produces a summary where Units and Revenue show totals (appropriate for quantities and money), while Profit shows both total and average (you want to know overall profit AND typical profit per transaction). The dictionary syntax gives you complete control over which aggregations apply to which columns.

2.3 Named Aggregations for Clean Results

The default column names from agg() can be confusing - you get multi-level column names like ('Profit', 'sum') and ('Profit', 'mean'). Named aggregations let you specify exactly what to call each result column.

```
# Named aggregations for clean, readable output
region_summary = sales.groupby('Region').agg(
    total_units=('Units', 'sum'),
    total_revenue=('Revenue', 'sum'),
    total_profit=('Profit', 'sum'),
    avg_profit=('Profit', 'mean'),
    transaction_count=('Units', 'count')
)
```

```
print("Region summary with named columns:")
print(region_summary)

# Now column names are meaningful and flat
print("\nColumn names:", list(region_summary.columns))
```

Named aggregations use the syntax `result_name=('column', 'function')`. The result is a clean DataFrame with flat, meaningful column names. This is the professional way to create summary tables - always use named aggregations when building reports or dashboards.

2.4 Part 2 Exercise

Exercise: Using the sales DataFrame from this section, create a product summary using named aggregations that shows: (1) total_units - sum of Units, (2) total_revenue - sum of Revenue, (3) avg_revenue - mean of Revenue, (4) min_profit - minimum Profit, (5) max_profit - maximum Profit. Group by Product and round the results to 2 decimal places.

3 Part 3: Custom Aggregation Functions

3.1 Lambda Functions for Custom Calculations

Sometimes built-in aggregations don't calculate what you need. Lambda functions let you define custom calculations inline. A lambda is a small anonymous function: `lambda x: expression` where x is the group's data.

```
# Calculate range (max - min) for each department's salaries
salary_range = employees.groupby('Department')['Salary'].agg(
    lambda x: x.max() - x.min()
)
print("Salary range by department:")
print(salary_range)

# Calculate coefficient of variation (std/mean * 100)
# This measures relative variability
cv = employees.groupby('Department')['Salary'].agg(
    lambda x: (x.std() / x.mean()) * 100
)
print("\nCoefficient of variation (%):")
print(cv.round(1))
```

The lambda receives x, which is a Series containing all values in that group. You can use any Series methods or operations on x. The salary range tells you the spread between lowest and highest paid employee. The coefficient of variation normalizes variability - a CV of 20% means the standard deviation is 20% of the mean, making it comparable across groups with different means.

3.2 Combining Built-in and Custom Aggregations

You can mix standard aggregations with custom lambda functions in the same agg() call. This combines the convenience of built-ins with the flexibility of custom calculations.

```
# Comprehensive salary analysis
salary_analysis = employees.groupby('Department')['Salary'].agg([
    'mean',
    'std',
    'min',
    'max',
    ('range', lambda x: x.max() - x.min()),
```

```
    ('cv_pct', lambda x: (x.std() / x.mean()) * 100)
])

print("Comprehensive salary analysis:")
print(salary_analysis.round(1))
```

When mixing built-ins and lambdas in a list, use tuples `('name', function)` for custom functions to give them meaningful names. The result is a complete statistical profile for each department in one clean table.

3.3 The apply() Method for Complex Operations

For operations that need access to multiple columns or complex logic, use `apply()` with a function that receives the entire group DataFrame. This is slower than agg() but more flexible.

```
# Function that calculates weighted average
def weighted_avg_salary(group):
    """Calculate salary weighted by years of experience"""
    # Simplified: weight senior positions more heavily
    weights = group['Level'].map({'Senior': 2, 'Junior': 1})
    return (group['Salary'] * weights).sum() / weights.sum()

# Apply to each department
weighted_salaries = employees.groupby('Department').apply(
    weighted_avg_salary
)
print("Weighted average salary by department:")
print(weighted_salaries)

# Compare to simple mean
simple_means = employees.groupby('Department')['Salary'].mean()
print("\nSimple mean for comparison:")
print(simple_means)
```

The function receives a DataFrame containing all rows for one group. You can access multiple columns, perform complex calculations, and return a single value. The weighted average gives Senior employees double weight, so departments with more Senior employees will have higher weighted averages than simple means.

3.4 Part 3 Exercise

Exercise: Create a custom aggregation that calculates the "interquartile range" (IQR = 75th percentile - 25th percentile) for salaries in each department. Use a lambda function with numpy's percentile function: `lambda x: np.percentile(x, 75) - np.percentile(x, 25)`. This measures the spread of the middle 50% of salaries, which is more robust to outliers than range.

4 Part 4: Filtering and Transforming Groups

4.1 Filtering Groups with filter()

Sometimes you want to keep or discard entire groups based on group-level criteria. The `filter()` method keeps groups where a condition is True and discards groups where it's False. The result maintains the original DataFrame structure but with some groups removed.

```
# Keep only departments with average salary > 70000
high_salary_depts = employees.groupby('Department').filter(
    lambda x: x['Salary'].mean() > 70000
)
```

```
print("Original employee count:", len(employees))
print("After filtering:", len(high_salary_depts))
print("\nEmployees in high-salary departments:")
print(high_salary_depts)

# Keep only departments with more than 3 employees
large_depts = employees.groupby('Department').filter(
    lambda x: len(x) > 3
)
print("\nEmployees in large departments:")
print(large_depts)
```

The filter function receives each group as a DataFrame and returns True/False. If the group's average salary exceeds $70,000, ALL employees in that department are kept. If not, ALL are removed. This is group-level filtering, not row-level - you're selecting entire groups based on their aggregate properties.

4.2 Transforming Groups with transform()

The `transform()` method applies a function to each group but returns results with the same shape as the input. This is perfect for adding group-level statistics back to individual rows - like showing each employee's salary compared to their department average.

```
# Add department mean salary to each employee's row
employees['Dept_Mean'] = employees.groupby('Department')['Salary'].transform('
    mean')

print("Employees with department mean:")
print(employees[['Name', 'Department', 'Salary', 'Dept_Mean']])

# Calculate how each employee compares to department average
employees['Vs_Dept_Avg'] = employees['Salary'] - employees['Dept_Mean']

print("\nWith comparison to department average:")
print(employees[['Name', 'Department', 'Salary', 'Vs_Dept_Avg']])
```

Transform broadcasts the group result back to each row in that group. Every Engineering employee gets the Engineering mean (77,250) in their Dept_Mean column. Now you can see that Alice in Sales earns $11,250 above her department average, while Bob earns $18,750 below. This individual-to-group comparison is extremely valuable for identifying outliers, high performers, or employees needing attention.

4.3 Practical Transform Applications

Transform is powerful for normalizing data within groups, calculating percentages of group totals, or creating group-based features for machine learning.

```
# Calculate each sale as percentage of regional total
sales['Pct_of_Region'] = sales.groupby('Region')['Revenue'].transform(
    lambda x: (x / x.sum()) * 100
)

print("Sales with regional percentage:")
print(sales[['Region', 'Product', 'Revenue', 'Pct_of_Region']].round(1))

# Normalize salaries within department (z-score)
employees['Salary_ZScore'] = employees.groupby('Department')['Salary'].
    transform(
    lambda x: (x - x.mean()) / x.std()
)
```

```
13
14 print("\nEmployees with z-score normalized salary:")
15 print(employees[['Name', 'Department', 'Salary', 'Salary_ZScore']].round(2))
```

The z-score normalization shows how many standard deviations each employee is from their department mean. A z-score of 1.0 means one standard deviation above average; -1.0 means one below. This makes salaries comparable across departments with different scales.

4.4 Part 4 Exercise

Exercise: Using the sales DataFrame: (1) Use filter() to keep only regions where total Revenue exceeds $3,000. (2) Use transform() to add a column showing each transaction's percentage of its region's total revenue. (3) Identify which specific transactions contribute more than 30% of their region's revenue.

5 Part 5: Pivot Tables

5.1 Understanding Pivot Tables

Pivot tables reshape data from long format (many rows, few columns) to wide format (fewer rows, more columns) while aggregating values. They're called pivot tables because you "pivot" one column's values to become column headers. This creates a matrix view that's perfect for comparing across two categorical dimensions.

Think of a spreadsheet where rows are regions and columns are products. Each cell shows total sales for that region-product combination. That's a pivot table - it takes your detailed transaction data and summarizes it in a grid that reveals patterns at a glance.

5.2 Creating Basic Pivot Tables

The `pivot_table()` function creates pivot tables from DataFrames. You specify which column's values become row indices (`index`), which become column headers (`columns`), which values to aggregate (`values`), and how to aggregate them (`aggfunc`).

```
1  # Create pivot table: Region (rows) x Product (columns)
2  # Values: Revenue, Aggregation: sum
3  pivot = pd.pivot_table(
4      sales,
5      values='Revenue',
6      index='Region',
7      columns='Product',
8      aggfunc='sum'
9  )
10
11 print("Revenue by Region and Product:")
12 print(pivot)
13
14 # Add row and column totals with margins=True
15 pivot_with_totals = pd.pivot_table(
16     sales,
17     values='Revenue',
18     index='Region',
19     columns='Product',
20     aggfunc='sum',
21     margins=True,
22     margins_name='Total'
23 )
24
25 print("\nWith totals:")
```

```
print(pivot_with_totals)
```

The pivot table immediately shows you that East sells more Gadgets ($2,250) while West sells more Widgets ($2,900). The margins add row totals (total per region) and column totals (total per product), plus a grand total. This matrix view makes comparisons trivial - you can scan across a row to compare products within a region, or down a column to compare regions for one product.

5.3 Multiple Aggregations in Pivot Tables

Pivot tables can calculate multiple aggregations simultaneously by passing a list to `aggfunc`. This creates a multi-level column structure with each aggregation as a sub-column.

```
# Multiple aggregations: sum and mean
pivot_multi = pd.pivot_table(
    sales,
    values='Revenue',
    index='Region',
    columns='Product',
    aggfunc=['sum', 'mean']
)

print("Sum and Mean Revenue:")
print(pivot_multi)

# Different aggregations for different columns
pivot_different = pd.pivot_table(
    sales,
    index='Region',
    columns='Product',
    aggfunc={
        'Revenue': 'sum',
        'Units': 'sum',
        'Profit': 'mean'
    }
)

print("\nDifferent aggregations per column:")
print(pivot_different)
```

The first pivot shows both total and average revenue for each region-product combination. The second shows totals for Revenue and Units, but average for Profit. This flexibility lets you build comprehensive summary matrices that answer multiple questions in one view.

5.4 Handling Missing Combinations

When not all combinations exist in your data (perhaps East never sold a particular product), pivot tables show NaN for missing combinations. Use `fill_value` to replace these with a meaningful default.

```
# Imagine some region-product combinations are missing
# fill_value replaces NaN with specified value
pivot_filled = pd.pivot_table(
    sales,
    values='Revenue',
    index='Region',
    columns='Product',
    aggfunc='sum',
    fill_value=0  # Missing combinations become 0
)

```

```
print("Pivot with missing values filled:")
print(pivot_filled)
```

Using `fill_value=0` is appropriate when missing means "no sales" (zero). In other contexts, you might use fill_value to indicate "not applicable" or leave NaN to honestly represent missing data.

5.5 Part 5 Exercise

Exercise: Create a pivot table from the sales DataFrame showing: (1) Rows: Region, (2) Columns: Product, (3) Values: both sum and mean of Revenue. Add margins for totals. Then answer: Which region-product combination has the highest total revenue? Which has the highest average revenue per transaction?

6 Part 6: Cross-Tabulation

6.1 Understanding Cross-Tabulation

Cross-tabulation (crosstab) creates frequency tables showing how often combinations of categorical values occur. While pivot tables aggregate numeric values, crosstabs count occurrences. They answer questions like "how many employees are in each department-level combination?" or "what's the distribution of grades across majors?"

```
# Count employees in each Department-Level combination
crosstab = pd.crosstab(
    employees['Department'],
    employees['Level']
)

print("Employee count by Department and Level:")
print(crosstab)

# Add margins for row/column totals
crosstab_margins = pd.crosstab(
    employees['Department'],
    employees['Level'],
    margins=True
)

print("\nWith totals:")
print(crosstab_margins)
```

The crosstab shows that Engineering has 2 Senior and 2 Junior employees, while Sales has 2 Senior and 2 Junior as well. The margins show totals: 4 employees per department, 4 Senior and 4 Junior overall. This is the simplest form of crosstab - just counting occurrences.

6.2 Normalized Cross-Tabulations

Raw counts are useful, but percentages often reveal patterns better. The `normalize` parameter converts counts to proportions: 'index' normalizes by row (each row sums to 1), 'columns' normalizes by column, and 'all' normalizes by grand total.

```
# Normalize by row (percentages within each department)
crosstab_row = pd.crosstab(
    employees['Department'],
    employees['Level'],
    normalize='index'
)

```

```
print("Percentage within each department:")
print((crosstab_row * 100).round(1))

# Normalize by column (percentages within each level)
crosstab_col = pd.crosstab(
    employees['Department'],
    employees['Level'],
    normalize='columns'
)

print("\nPercentage within each level:")
print((crosstab_col * 100).round(1))
```

Row normalization shows that Engineering is 50% Senior and 50% Junior. Column normalization shows that among all Senior employees, 50% are in Engineering and 50% in Sales. These different normalizations answer different questions: "What's the seniority distribution within each department?" vs "How are senior employees distributed across departments?"

6.3 Cross-Tabulation with Aggregation

Crosstabs can also aggregate values like pivot tables by specifying `values` and `aggfunc`. This combines the frequency table structure with numeric aggregation.

```
# Average salary for each Department-Level combination
crosstab_agg = pd.crosstab(
    employees['Department'],
    employees['Level'],
    values=employees['Salary'],
    aggfunc='mean'
)

print("Average salary by Department and Level:")
print(crosstab_agg)

# This is equivalent to pivot_table but sometimes more concise
```

When you add values and aggfunc, crosstab behaves like pivot_table. Use whichever syntax feels more natural for your analysis - they achieve the same result.

6.4 Part 6 Exercise

Exercise: Using the employees DataFrame: (1) Create a crosstab showing the count of employees by Department and Level. (2) Normalize by 'index' to show percentages within each department. (3) Create another crosstab that shows average Salary for each Department-Level combination. Compare this with a pivot_table that produces the same result.

7 Part 7: Complete Analysis Workflow

7.1 Real-World Scenario: Sales Performance Analysis

Let's build a complete analysis workflow that demonstrates how these techniques combine. The scenario: you're analyzing regional sales performance to identify top performers, compare regions, and find improvement opportunities.

```
import pandas as pd
import numpy as np

# Create comprehensive sales dataset
np.random.seed(42)
```

```
sales_data = pd.DataFrame({
    'Date': pd.date_range('2024-01-01', periods=100, freq='D'),
    'Region': np.random.choice(['East', 'West', 'North', 'South'], 100),
    'Product': np.random.choice(['Widget', 'Gadget', 'Tool'], 100),
    'Salesperson': np.random.choice(['Alice', 'Bob', 'Charlie',
                                     'Diana', 'Eve'], 100),
    'Units': np.random.randint(10, 100, 100),
    'Revenue': np.random.randint(500, 5000, 100)
})

# Calculate profit (20-30% of revenue)
sales_data['Profit'] = (sales_data['Revenue'] *
                        np.random.uniform(0.2, 0.3, 100)).round(2)

print("Sales data sample:")
print(sales_data.head(10))
print(f"\nTotal records: {len(sales_data)}")
```

We created a realistic sales dataset with dates, regions, products, salespeople, units, revenue, and profit. Now let's analyze it systematically.

7.2 Step 1: Regional Performance Summary

```
# Comprehensive regional analysis with named aggregations
regional_summary = sales_data.groupby('Region').agg(
    total_revenue=('Revenue', 'sum'),
    avg_revenue=('Revenue', 'mean'),
    total_units=('Units', 'sum'),
    total_profit=('Profit', 'sum'),
    profit_margin=('Profit', lambda x: (x.sum() /
                   sales_data.loc[x.index, 'Revenue'].sum() * 100)),
    transaction_count=('Revenue', 'count')
).round(2)

print("Regional Performance Summary:")
print(regional_summary.sort_values('total_revenue', ascending=False))
```

This summary shows each region's total revenue, average transaction size, total units, profit, profit margin percentage, and transaction count. Sorting by total revenue identifies the top-performing region.

7.3 Step 2: Product Performance by Region (Pivot Table)

```
# Create pivot table for region-product analysis
product_pivot = pd.pivot_table(
    sales_data,
    values='Revenue',
    index='Region',
    columns='Product',
    aggfunc='sum',
    margins=True,
    fill_value=0
)

print("\nRevenue by Region and Product:")
print(product_pivot)

# Which product performs best in each region?
best_product = product_pivot.drop('All').idxmax(axis=1)
print("\nBest-selling product by region:")
print(best_product)
```

The pivot table reveals regional product preferences. Perhaps Widgets sell best in the East while Gadgets dominate the West. This informs inventory allocation and marketing strategies.

7.4 Step 3: Salesperson Performance with Transform

```
# Add regional context to individual sales
sales_data['Regional_Avg'] = sales_data.groupby('Region')['Revenue'].transform('mean')
sales_data['Vs_Regional_Avg'] = sales_data['Revenue'] - sales_data['Regional_Avg']

# Identify above-average transactions
above_avg = sales_data[sales_data['Vs_Regional_Avg'] > 0]
print(f"\nTransactions above regional average: {len(above_avg)} of {len(sales_data)}")

# Salesperson performance relative to region
salesperson_performance = sales_data.groupby('Salesperson').agg(
    total_revenue=('Revenue', 'sum'),
    avg_above_regional=('Vs_Regional_Avg', 'mean'),
    transaction_count=('Revenue', 'count')
).round(2)

print("\nSalesperson Performance:")
print(salesperson_performance.sort_values('avg_above_regional', ascending=False))
```

Transform added regional context to each transaction. Now we can see which salespeople consistently sell above their regional average - these are truly high performers, not just people assigned to high-revenue regions.

7.5 Step 4: Cross-Tabulation for Distribution Analysis

```
# How are salespeople distributed across regions?
salesperson_distribution = pd.crosstab(
    sales_data['Region'],
    sales_data['Salesperson'],
    normalize='index'
)

print("\nSalesperson distribution by region (%):")
print((salesperson_distribution * 100).round(1))

# Are sales evenly distributed across products?
product_distribution = pd.crosstab(
    sales_data['Region'],
    sales_data['Product'],
    values=sales_data['Revenue'],
    aggfunc='sum',
    normalize='index'
)

print("\nRevenue distribution by product within regions (%):")
print((product_distribution * 100).round(1))
```

These crosstabs reveal whether resources are balanced. If one salesperson handles 50% of a region's transactions, they might be overloaded. If one product accounts for 70% of a region's revenue, there might be diversification opportunities.

7.6 Step 5: Save Analysis Results

```
# Save all analysis results for reporting
regional_summary.to_csv('regional_performance.csv')
product_pivot.to_csv('product_by_region.csv')
salesperson_performance.to_csv('salesperson_performance.csv')

print("\nAnalysis complete! Results saved to CSV files:")
print("- regional_performance.csv")
print("- product_by_region.csv")
print("- salesperson_performance.csv")
```

The complete workflow demonstrates how aggregation techniques combine: named aggregations for clean summaries, pivot tables for cross-dimensional analysis, transform for contextual comparisons, and crosstabs for distribution analysis. This is how professional data analysts approach real business questions.

7.7 Part 7 Exercise

Exercise: Using the sales_data from this workflow: (1) Add a 'Quarter' column based on the Date. (2) Create a complete quarterly analysis that includes: regional revenue summary with named aggregations, a pivot table of revenue by Region and Quarter, transform to show each sale's percentage of its quarterly total, and a crosstab showing transaction counts by Region and Product. (3) Save all results to CSV files.

Summary and Best Practices

Key Takeaways:

1. Multi-column groupby creates groups for each unique combination of values across multiple categorical dimensions
2. Use agg() with lists for multiple aggregations on one column, dictionaries for different aggregations per column
3. Named aggregations provide clean, readable column names - always use them for reports
4. Lambda functions enable custom calculations like range, coefficient of variation, or weighted averages
5. filter() keeps or discards entire groups based on group-level criteria
6. transform() broadcasts group statistics back to individual rows, perfect for comparisons and normalizations
7. pivot_table() reshapes data into matrix form for cross-dimensional analysis
8. crosstab() creates frequency tables and can normalize to show percentages
9. Combine techniques in workflows: summarize, pivot, transform, then save results

Common Pitfalls to Avoid

- Forgetting to reset_index() after multi-column groupby, making results hard to work with
- Using wrong normalization in crosstab ('index' vs 'columns' vs 'all')

- Confusing filter() (keeps/discards groups) with boolean indexing (keeps/discards rows)
- Using apply() when agg() would work - apply() is slower and should be reserved for complex operations
- Creating pivot tables without fill_value when missing combinations should be zeros
- Forgetting that transform() returns same shape as input while agg() returns summary
- Not naming aggregations, resulting in confusing multi-level column names
- Interpreting group statistics without checking group sizes - small groups give unreliable statistics

Next Lecture Preview

In Lecture 21, we'll learn data visualization with Matplotlib and Seaborn. You'll transform your analytical insights into compelling visual stories - creating line plots for trends, bar charts for comparisons, scatter plots for relationships, and histograms for distributions. The aggregation skills you've mastered today produce the summarized data that visualizations display. You'll learn to take a pivot table and turn it into a heatmap, or take a groupby result and create a grouped bar chart. Visualization is the final step in the data analysis workflow - communicating your findings clearly and effectively.