

# Data Manipulation Challenge

## A Mental Model for Method Chaining in Pandas

### Data Manipulation Challenge - A Mental Model for Method Chaining in Pandas

! Challenge Requirements In Section [Student Analysis Section](#)

- Complete all discussion questions for the seven mental models (plus some extra requirements for higher grades)

! Note on Python Usage

**Recommended Workflow: Use Your Existing Virtual Environment** If you completed the Tech Setup Challenge Part 2, you already have a virtual environment set up! Here's how to use it for this new challenge:

1. **Clone this new challenge repository** (see Getting Started section below)
2. **Open the cloned repository in Cursor**
3. **Set this project to use your existing Python interpreter:**
  - Press **Ctrl+Shift+P** → “Python: Select Interpreter”
  - Navigate to and choose the interpreter from your existing virtual environment (e.g., `your-previous-project/venv/Scripts/python.exe`)
4. **Activate the environment in your terminal:**
  - Open terminal in Cursor (‘Ctrl + ‘)
  - Navigate to your previous project folder where you have the `venv` folder
  - **Pro tip:** You can quickly navigate by typing `cd` followed by dragging the folder from your file explorer into the terminal
  - Activate using the appropriate command for your system:
    - **Windows Command Prompt:** `venv\Scripts\activate`

- **Windows PowerShell:** `.\venv\Scripts\Activate.ps1`
- **Mac/Linux:** `source venv/bin/activate`
- You should see `(venv)` at the beginning of your terminal prompt

5. **Install additional packages if needed:** `pip install pandas numpy matplotlib seaborn`

#### Cloud Storage Warning

**Avoid using Google Drive, OneDrive, or other cloud storage for Python projects!** These services can cause issues with: - Package installations failing due to file locking - Virtual environment corruption - Slow performance during pip operations

**Best practice:** Keep your Python projects in a local folder like `C:\Users\YourName\Documents\` or `~/Documents/` instead of cloud-synced folders.

**Alternative: Create a New Virtual Environment** If you prefer a fresh environment, follow the Quarto documentation: <https://quarto.org/docs/projects/virtual-environments.html>. Be sure to follow the instructions to activate the environment, set it up as your default Python interpreter for the project, and install the necessary packages (e.g. pandas) for this challenge. For installing the packages, you can use the `pip install -r requirements.txt` command since you already have the `requirements.txt` file in your project. Some steps do take a bit of time, so be patient.

**Why This Works:** Virtual environments are portable - you can use the same environment across multiple projects, and Cursor automatically activates it when you select the interpreter!

## The Problem: Mastering Data Manipulation Through Method Chaining

**Core Question:** How can we efficiently manipulate datasets using `pandas` method chaining to answer complex business questions?

**The Challenge:** Real-world data analysis requires combining multiple data manipulation techniques in sequence. Rather than creating intermediate variables at each step, method chaining allows us to write clean, readable code that flows logically from one operation to the next.

**Our Approach:** We'll work with ZappTech's shipment data to answer critical business questions about service levels and cross-category orders, using the seven mental models of data manipulation through `pandas` method chaining.

### AI Partnership Required

This challenge pushes boundaries intentionally. You'll tackle problems that normally require weeks of study, but with Cursor AI as your partner (and your brain keeping it honest), you can accomplish more than you thought possible.

**The new reality:** The four stages of competence are Ignorance → Awareness → Learning → Mastery. AI lets us produce Mastery-level work while operating primarily in the Awareness stage. I focus on awareness training, you leverage AI for execution, and together we create outputs that used to require years of dedicated study.

## The Seven Mental Models of Data Manipulation

The seven most important ways we manipulate datasets are:

1. **Assign:** Add new variables with calculations and transformations
2. **Subset:** Filter data based on conditions or select specific columns
3. **Drop:** Remove unwanted variables or observations
4. **Sort:** Arrange data by values or indices
5. **Aggregate:** Summarize data using functions like mean, sum, count
6. **Merge:** Combine information from multiple datasets
7. **Split-Apply-Combine:** Group data and apply functions within groups

## Data and Business Context

We analyze ZappTech's shipment data, which contains information about product deliveries across multiple categories. This dataset is ideal for our analysis because:

- **Real Business Questions:** CEO wants to understand service levels and cross-category shopping patterns
- **Multiple Data Sources:** Requires merging shipment data with product category information
- **Complex Relationships:** Service levels may vary by product category, and customers may order across categories
- **Method Chaining Practice:** Perfect for demonstrating all seven mental models in sequence

## Data Loading and Initial Exploration

Let's start by loading the ZappTech shipment data and understanding what we're working with.

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta

# Load the shipment data
shipments_df = pd.read_csv(
    "https://raw.githubusercontent.com/flyaflya/persuasive/main/shipments.csv",
    parse_dates=['plannedShipDate', 'actualShipDate']
)

# Load product line data
product_line_df = pd.read_csv(
    "https://raw.githubusercontent.com/flyaflya/persuasive/main/productLine.csv"
)

# Reduce dataset size for faster processing (4,000 rows instead of 96,805 rows)
shipments_df = shipments_df.head(4000)

print("Shipments data shape:", shipments_df.shape)
print("\nShipments data columns:", shipments_df.columns.tolist())
print("\nFirst few rows of shipments data:")
print(shipments_df.head(10))

print("\n" + "="*50)
print("Product line data shape:", product_line_df.shape)
print("\nProduct line data columns:", product_line_df.columns.tolist())
print("\nFirst few rows of product line data:")
print(product_line_df.head(10))

```

Shipments data shape: (4000, 5)

Shipments data columns: ['shipID', 'plannedShipDate', 'actualShipDate', 'partID', 'quantity']

First few rows of shipments data:

|   | shipID | plannedShipDate | actualShipDate | partID      | quantity |
|---|--------|-----------------|----------------|-------------|----------|
| 0 | 10001  | 2013-11-06      | 2013-10-04     | part92b16c5 | 6        |
| 1 | 10002  | 2013-10-15      | 2013-10-04     | part66983b  | 2        |
| 2 | 10003  | 2013-10-25      | 2013-10-07     | part8e36f25 | 1        |
| 3 | 10004  | 2013-10-14      | 2013-10-08     | part30f5de0 | 1        |
| 4 | 10005  | 2013-10-14      | 2013-10-08     | part9d64d35 | 6        |

|   |       |            |            |             |    |
|---|-------|------------|------------|-------------|----|
| 5 | 10006 | 2013-10-14 | 2013-10-08 | part6cd6167 | 15 |
| 6 | 10007 | 2013-10-14 | 2013-10-08 | parta4d5fd1 | 2  |
| 7 | 10008 | 2013-10-14 | 2013-10-08 | part08cadf5 | 1  |
| 8 | 10009 | 2013-10-14 | 2013-10-08 | part5cc4989 | 10 |
| 9 | 10010 | 2013-10-14 | 2013-10-08 | part912ae4c | 1  |

=====

Product line data shape: (11997, 3)

Product line data columns: ['partID', 'productLine', 'prodCategory']

First few rows of product line data:

|   | partID      | productLine | prodCategory |
|---|-------------|-------------|--------------|
| 0 | part00005ba | line4c      | Liquids      |
| 1 | part000b57d | line61      | Machines     |
| 2 | part00123bf | linec1      | Marketable   |
| 3 | part0021fc9 | line61      | Machines     |
| 4 | part0027e86 | line2f      | Machines     |
| 5 | part002ed95 | line4c      | Liquids      |
| 6 | part0030856 | lineb8      | Machines     |
| 7 | part0033dfd | line49      | Liquids      |
| 8 | part0037a2a | linea3      | Marketable   |
| 9 | part003caee | linea3      | Marketable   |

### Understanding the Data

**Shipments Data:** Contains individual line items for each shipment, including: - **shipID:** Unique identifier for each shipment - **partID:** Product identifier - **plannedShipDate:** When the shipment was supposed to go out - **actualShipDate:** When it actually shipped - **quantity:** How many units were shipped

**Product Category and Line Data:** Contains product category information: - **partID:** Links to shipments data - **productLine:** The category each product belongs to - **prodCategory:** The category each product belongs to

**Business Questions We'll Answer:** 1. Does service level (on-time shipments) vary across product categories? 2. How often do orders include products from more than one category?

## The Seven Mental Models: A Progressive Learning Journey

Now we'll work through each of the seven mental models using method chaining, starting simple and building complexity.

## 1. Assign: Adding New Variables

**Mental Model:** Create new columns with calculations and transformations.

Let's start by calculating whether each shipment was late:

```
# Simple assignment - calculate if shipment was late
shipments_with_lateness = (
    shipments_df
    .assign(
        is_late=lambda df: df['actualShipDate'] > df['plannedShipDate'],
        days_late=lambda df: (df['actualShipDate'] - df['plannedShipDate']).dt.days
    )
)

print("Added lateness calculations:")
print(shipments_with_lateness[['shipID', 'plannedShipDate', 'actualShipDate', 'is_late', 'days_late']])
```

Added lateness calculations:

|   | shipID | plannedShipDate | actualShipDate | is_late | days_late |
|---|--------|-----------------|----------------|---------|-----------|
| 0 | 10001  | 2013-11-06      | 2013-10-04     | False   | -33       |
| 1 | 10002  | 2013-10-15      | 2013-10-04     | False   | -11       |
| 2 | 10003  | 2013-10-25      | 2013-10-07     | False   | -18       |
| 3 | 10004  | 2013-10-14      | 2013-10-08     | False   | -6        |
| 4 | 10005  | 2013-10-14      | 2013-10-08     | False   | -6        |

### Method Chaining Tip for New Python Users

**Why use `lambda df:`?** When chaining methods, we need to reference the current state of the dataframe. The `lambda df:` tells pandas “use the current dataframe in this calculation.” Without it, pandas would look for a variable called `df` that doesn’t exist.

**Alternative approach:** You could also write this as separate steps, but method chaining keeps related operations together and makes the code more readable.

### Discussion Questions: Assign Mental Model

**Question 1: Data Types and Date Handling** - What is the `dtype` of the `actualShipDate` series? How can you find out using code? - Why is it important that both `actualShipDate` and `plannedShipDate` have the same data type for comparison?

**Question 2: String vs Date Comparison** - Can you give an example where comparing two dates as strings would yield unintuitive results, e.g. what happens if you try to compare “04-11-2025” and “05-20-2024” as strings vs as dates?

### Question 3: Debug This Code

```
# This code has an error - can you spot it?
shipments_with_lateness = (
    shipments_df
    .assign(
        is_late=lambda df: df['actualShipDate'] > df['plannedShipDate'],
        days_late=lambda df: (df['actualShipDate'] - df['plannedShipDate']).dt.days,
        lateStatement="Darn Shipment is Late" if shipments_df['is_late'] else "Shipment is on time"
    )
)
```

What's wrong with the `lateStatement` assignment and how would you fix it?

### Briefly Give Answers to the Discussion Questions In This Section

---

#### Question 1: Data Types and Date Handling

##### Data Type of Dates

The `actualShipDate` column is a `datetime64[ns]` type - a specialized format that treats each value as a proper calendar date with time information. You can check this using the `.dtype` attribute on any pandas Series.

##### Why Same Types Matter

Think of it like sorting apples and oranges:

##### Note

If one date is text ("2025-04-11") and another is a real date object:

- The computer won't understand they're dates
- It can't calculate time differences
- It can't properly compare which came first

#### Question 2: String vs Date Comparison

### ! The Problem with String Dates

When comparing “04-11-2025” and “05-20-2024” as strings:

- The computer looks at characters from left to right
- It sees “04” < “05”, so thinks April 2025 comes before May 2024
- This is wrong! 2025 should come after 2024

### 💡 Real-World Analogy

It’s like sorting phone numbers as text: - As text: “2” comes after “11” (wrong!) - As numbers: 2 comes before 11 (correct!)

Same with dates - they need to be proper date objects for correct ordering.

## Question 3: Debug This Code

### ! What’s Wrong

The code tries to use a Python if/else statement inside `.assign()`, but:

1. It breaks the column-wise operation concept
2. It tries to access `shipments_df['is_late']` which doesn’t exist yet
3. The comma after the string makes it invalid syntax

### 💡 How to Fix It

Use a lambda function with `np.where` to handle the if/else:

```
shipments_with_lateness = (  
    shipments_df  
    .assign(  
        is_late=lambda df: df['actualShipDate'] > df['plannedShipDate'],  
        days_late=lambda df: (df['actualShipDate'] - df['plannedShipDate']).dt.days,  
        lateStatement=lambda df: np.where(df['is_late'],  
                                           "Darn Shipment is Late",  
                                           "Shipment is on Time")  
    )  
)
```



## 2. Subset: Querying Rows and Filtering Columns

**Mental Model:** Query rows based on conditions and filter to keep specific columns.

Let's query for only late shipments and filter to keep the columns we need:

```
# Query rows for late shipments and filter to keep specific columns
late_shipments = (
    shipments_with_lateness
    .query('is_late == True') # Query rows where is_late is True
    .filter(['shipID', 'partID', 'plannedShipDate', 'actualShipDate', 'days_late']) # Filter
)

print(f"Found {len(late_shipments)} late shipments out of {len(shipments_with_lateness)} total")
print("\nLate shipments sample:")
print(late_shipments.head())
```

Found 456 late shipments out of 4000 total

Late shipments sample:

|     | shipID | partID      | plannedShipDate | actualShipDate | days_late |
|-----|--------|-------------|-----------------|----------------|-----------|
| 776 | 10192  | part0164a70 | 2013-10-09      | 2013-10-14     | 5         |
| 777 | 10192  | part9259836 | 2013-10-09      | 2013-10-14     | 5         |
| 778 | 10192  | part4526c73 | 2013-10-09      | 2013-10-14     | 5         |
| 779 | 10192  | partbb47e81 | 2013-10-09      | 2013-10-14     | 5         |
| 780 | 10192  | part008482f | 2013-10-09      | 2013-10-14     | 5         |

### Understanding the Methods

- **.query()**: Query rows based on conditions (like SQL WHERE clause)
- **.filter()**: Filter to keep specific columns by name
- **Alternative:** You could use **.loc[]** for more complex row querying, but **.query()** is often more readable

### Discussion Questions: Subset Mental Model

**Question 1: Query vs Boolean Indexing** - What's the difference between using **.query('is\_late == True')** and **[df['is\_late'] == True]**? - Which approach is more readable and why?

**Question 2: Additional Row Querying** - Can you show an example of using a variable like **late\_threshold** to query rows for shipments that are at least **late\_threshold** days late, e.g. what if you wanted to query rows for shipments that are at least 5 days late?

## Briefly Give Answers to the Discussion Questions In This Section

---

### Question 1: Query vs Boolean Indexing

- `.query('is_late == True')` reads like a sentence and is easy to scan. It's good for simple conditions and keeps code tidy.
- `df[df['is_late'] == True]` is explicit and flexible: you can build complex conditions with normal Python operators and variables.

Which is more readable? - For short, simple conditions, `.query()` is usually more readable because it looks like plain English. - For complex logic or when you need to use Python variables directly, boolean indexing (`df[...]`) is clearer.

### Question 2: Using a `late_threshold`

If you want shipments at least `late_threshold` days late, define the variable and use boolean indexing. This is clear and flexible.

Example: - Set `late_threshold` to a number (for example, 5) - Keep rows where `days_late` is greater than or equal to `late_threshold`

Why use boolean indexing here? - It makes it simple to plug in a variable - The condition can use any combined logic (AND, OR, parentheses)

---

## 3. Drop: Removing Unwanted Data

**Mental Model:** Remove columns or rows you don't need.

Let's clean up our data by removing unnecessary columns:

```
# Create a cleaner dataset by dropping unnecessary columns
clean_shipments = (
    shipments_with_lateness
    .drop(columns=['quantity']) # Drop quantity column (not needed for our analysis)
    .dropna(subset=['plannedShipDate', 'actualShipDate']) # Remove rows with missing dates
)

print(f"Cleaned dataset: {len(clean_shipments)} rows, {len(clean_shipments.columns)} columns")
print("Remaining columns:", clean_shipments.columns.tolist())
```

Cleaned dataset: 4000 rows, 6 columns

Remaining columns: ['shipID', 'plannedShipDate', 'actualShipDate', 'partID', 'is\_late', 'days']

### ! Discussion Questions: Drop Mental Model

**Question 1: Drop vs Filter Strategies** - What's the difference between `.drop(columns=['quantity'])` and `.filter()` with a list of columns you want to keep? - When would you choose to drop columns vs filter to keep specific columns?

**Question 2: Handling Missing Data** - What happens if you use `.dropna()` without specifying `subset`? How is this different from `.dropna(subset=['plannedShipDate', 'actualShipDate'])`? - Why might you want to be selective about which columns to check for missing values?

## Briefly Give Answers to the Discussion Questions In This Section

---

### Question 1: Drop vs Filter Strategies

#### i Key Difference

- `.drop()` removes specified columns
- `.filter()` keeps specified columns

It's like two ways to clean your desk: - **drop**: "Remove these three items" - **filter**: "Keep only these five items"

#### 💡 When to Use Each

Choose `.drop()` when: - You have a few unwanted columns - You want to preserve most columns - You're removing problematic columns

Choose `.filter()` when: - You know exactly which columns you need - You want to create a focused subset - You're building a specific view of the data

### Question 2: Missing Data Handling

#### ! Understanding `.dropna()`

- Without `subset`: Removes rows with ANY missing value
- With `subset`: Only checks specified columns

Think of it like attendance: - Without subset: “Remove if student missed ANY class” -  
With subset: “Remove only if missed MATH or SCIENCE”

### **i** Why Be Selective?

Use **subset** when: - Some columns can have missing values (like optional fields) - Critical calculations need complete data - You want to preserve more rows

Example: In shipping data - Missing dates are critical (can't calculate delays) - Missing product notes are okay (not needed for timing analysis)

---

## 4. Sort: Arranging Data

**Mental Model:** Order data by values or indices.

Let's sort by lateness to see the worst offenders:

```
# Sort by days late (worst first)
sorted_by_lateness = (
    clean_shipments
    .sort_values('days_late', ascending=False) # Sort by days_late, highest first
    .reset_index(drop=True) # Reset index to be sequential
)

print("Shipments sorted by lateness (worst first):")
print(sorted_by_lateness[['shipID', 'partID', 'days_late', 'is_late']].head(10))
```

Shipments sorted by lateness (worst first):

|   | shipID | partID      | days_late | is_late |
|---|--------|-------------|-----------|---------|
| 0 | 10956  | partc63f9bc | 21        | True    |
| 1 | 10956  | part3017fa1 | 21        | True    |
| 2 | 10956  | part1fedfcf | 21        | True    |
| 3 | 10956  | parte820e31 | 21        | True    |
| 4 | 10956  | partb6208b5 | 21        | True    |
| 5 | 10956  | partc979912 | 21        | True    |
| 6 | 10956  | partc653823 | 21        | True    |
| 7 | 10956  | partb722d53 | 21        | True    |
| 8 | 10956  | part4875f85 | 21        | True    |
| 9 | 10956  | part04ef2f7 | 21        | True    |

## ! Discussion Questions: Sort Mental Model

**Question 1: Sorting Strategies** - What's the difference between `ascending=False` and `ascending=True` in sorting? - How would you sort by multiple columns (e.g., first by `is_late`, then by `days_late`)?

**Question 2: Index Management** - Why do we use `.reset_index(drop=True)` after sorting? - What happens to the original index when you sort? Why might this be problematic?

## Briefly Give Answers to the Discussion Questions In This Section

---

### Question 1: Sorting Strategies

**i** `ascending=True` vs `False`

- `ascending=True`: A→Z, 1→9 (smallest first)
- `ascending=False`: Z→A, 9→1 (largest first)

In our shipping data: - `ascending=True`: Show on-time shipments first - `ascending=False`: Show most delayed shipments first

**💡** Sorting by Multiple Columns

Use a list of columns and a list of directions:

```
df.sort_values(  
    by=['is_late', 'days_late'],  
    ascending=[True, False]  
)
```

1. First by section (`is_late`)
2. Then by publication date (`days_late`)

### Question 2: Index Management

### ! Why Reset Index?

The original index is like sticky notes on the data:

- When we sort, the sticky notes stay with their original rows
- This breaks the visual “1,2,3...” sequence
- `.reset_index(drop=True)` gives fresh, sequential numbers

### ⚠ Problems with Original Index

Without resetting: - Row 5 might appear before Row 1 - Position Index number - Hard to reference “first N rows”

Example from shipping data:

| Index | days_late |
|-------|-----------|
| 20    | 5         |
| 3     | 4         |
| 15    | 3         |

Confusing! Are we looking at row 1, 2, or 3?

---

## 5. Aggregate: Summarizing Data

**Mental Model:** Calculate summary statistics across groups or the entire dataset.

Let’s calculate overall service level metrics:

```
# Calculate overall service level metrics
service_metrics = (
    clean_shipments
    .agg({
        'is_late': ['count', 'sum', 'mean'], # Count total, count late, calculate percentage
        'days_late': ['mean', 'max'] # Average and maximum days late
    })
    .round(3)
)

print("Overall Service Level Metrics:")
print(service_metrics)
```

```
# Calculate percentage on-time directly from the data
on_time_rate = (1 - clean_shipments['is_late'].mean()) * 100
print(f"\nOn-time delivery rate: {on_time_rate:.1f}%")
```

Overall Service Level Metrics:

|       | is_late  | days_late |
|-------|----------|-----------|
| count | 4000.000 | NaN       |
| sum   | 456.000  | NaN       |
| mean  | 0.114    | -0.974    |
| max   | NaN      | 21.000    |

On-time delivery rate: 88.6%

### ! Discussion Questions: Aggregate Mental Model

**Question 1: Boolean Aggregation** - Why does `sum()` work on boolean values? What does it count?

**Briefly Give Answers to the Discussion Questions In This Section**

---

#### Question 1: Boolean Aggregation

**i** Why does `sum()` work on boolean values?

In Python, boolean values are a subclass of integers where:

- `True` is equivalent to 1
- `False` is equivalent to 0

**i** Question 2: What does it count?\*

When you use `sum()` on a boolean column:

- It counts the number of `True` values
  - In this case, it's counting the total number of late shipments since `is_late` is `True` for late shipments
-

## 6. Merge: Combining Information

**Mental Model:** Join data from multiple sources to create richer datasets.

Now let's analyze service levels by product category. First, we need to merge our data:

```
# Merge shipment data with product line data
shipments_with_category = (
    clean_shipments
    .merge(product_line_df, on='partID', how='left') # Left join to keep all shipments
    .assign(
        category_late=lambda df: df['is_late'] & df['prodCategory'].notna() # Only count as
    )
)

print("\nProduct categories available:")
print(shipments_with_category['prodCategory'].value_counts())
```

```
Product categories available:
prodCategory
Marketable      1850
Machines        846
SpareParts      767
Liquids         537
Name: count, dtype: int64
```

### ! Discussion Questions: Merge Mental Model

**Question 1: Join Types and Data Loss** - Why does your professor think we should use `how='left'` in most cases? - How can you check if any shipments were lost during the merge?

**Question 2: Key Column Matching** - What happens if there are duplicate `partID` values in the `product_line_df`?

**Briefly Give Answers to the Discussion Questions In This Section**

---

**Question 1: Join Types and Data Loss**



**i** Why use how='left' in most cases?

The recommendation to use how='left' in most joins is based on several important considerations:

**1. Data Preservation:**

- Left joins preserve ALL records from your primary (left) dataset
- This ensures no data loss from your main analysis dataset
- Critical when analyzing business metrics like shipment performance

**2. Audit Trail:**

- Left joins make it easy to identify missing matches
- You can spot null values in the joined columns
- Helps maintain data quality control

**i** How to check for lost shipments during merge?

**1. We can check for data loss in joins using these methods:**

- Compare row counts: Length of Original shipments df and After left join
- Identify unmatched records: Find partIDs that exist in shipments but not in product\_line

## Question 2: Key Column Matching

**i** What happens if there are duplicate partID values in the product\_line\_df?

When there are duplicate partID values in product\_line\_df, the join behavior becomes more complex:

**1. Data Multiplication:**

- Each matching record in the left dataset will be paired with ALL matching records in the right dataset
- This can lead to unexpected row multiplication

---

## 7. Split-Apply-Combine: Group Analysis

**Mental Model:** Group data and apply functions within each group.

Now let's analyze service levels by category:

```
# Analyze service levels by product category
service_by_category = (
    shipments_with_category
    .groupby('prodCategory') # Split by product category
    .agg({
        'is_late': ['any', 'count', 'sum', 'mean'], # Count, late count, percentage late
        'days_late': ['mean', 'max'] # Average and max days late
    })
    .round(3)
)

print("Service Level by Product Category:")
print(service_by_category)
```

Service Level by Product Category:

| prodCategory | is_late |       |     | days_late |        |     |
|--------------|---------|-------|-----|-----------|--------|-----|
|              | any     | count | sum | mean      | mean   | max |
| Liquids      | True    | 537   | 22  | 0.041     | -0.950 | 19  |
| Machines     | True    | 846   | 152 | 0.180     | -1.336 | 21  |
| Marketable   | True    | 1850  | 145 | 0.078     | -0.804 | 21  |
| SpareParts   | True    | 767   | 137 | 0.179     | -1.003 | 21  |

### ! Discussion Questions: Split-Apply-Combine Mental Model

**Question 1: GroupBy Mechanics** - What does `.groupby('prodCategory')` actually do? How does it “split” the data? - Why do we need to use `.agg()` after grouping? What happens if you don't?

**Question 2: Multi-Level Grouping** - Explore grouping by `['shipID', 'prodCategory']`? What question does this answer versus grouping by `'prodCategory'` alone? (HINT: There may be many rows with identical `shipID`'s due to a particular order having multiple `partID`'s.)

**Briefly Give Answers to the Discussion Questions In This Section**

---

**Question 1: GroupBy Mechanics**

**i** What does `.groupby('prodCategory')` actually do?

- It splits the **DataFrame** into **subgroups** based on unique values in the `prodCategory` column.

**i** How does it “split” the data?

- Internally, it creates a dictionary-like structure where each key is a unique `prodCategory`, and the value is the subset of rows belonging to that category.

**i** Why do we need to use `.agg()` after grouping? What happens if you don't?

- `.agg()` tells pandas **what to compute** for each group — like **mean**, **sum**, **count**, etc.
- Without `.agg()`, we just have a grouped object — no calculations are performed.

## Question 2: Multi-Level Grouping

**i** Exploring grouping by `['shipID', 'prodCategory']`

- It creates **hierarchical groups** based on **combinations** of `shipID` and `prodCategory`.
- Each group represents a unique pairing — e.g., shipment **S123** with category **Electronics**.

**i** What question does this answer versus grouping by `'prodCategory'` alone?

- How did each shipment perform **across** different **product categories**?
- Captures cases where a single **shipID** includes multiple **partIDs** from different categories.

---

## Answering A Business Question

**Mental Model:** Combine multiple data manipulation techniques to answer complex business questions.

Let's create a comprehensive analysis by combining shipment-level data with category information:

```

# Create a comprehensive analysis dataset
comprehensive_analysis = (
    shipments_with_category
    .groupby(['shipID', 'prodCategory']) # Group by shipment and category
    .agg({
        'is_late': 'any', # True if any item in this shipment/category is late
        'days_late': 'max' # Maximum days late for this shipment/category
    })
    .reset_index()
    .assign(
        has_multiple_categories=lambda df: df.groupby('shipID')['prodCategory'].transform('n
    )
)

print("Comprehensive analysis - shipments with multiple categories:")
multi_category_shipments = comprehensive_analysis[comprehensive_analysis['has_multiple_categ
print(f"Shipments with multiple categories: {multi_category_shipments['shipID'].nunique()}")
print(f"Total unique shipments: {comprehensive_analysis['shipID'].nunique()}")
print(f"Percentage with multiple categories: {multi_category_shipments['shipID'].nunique() /

```

Comprehensive analysis - shipments with multiple categories:  
Shipments with multiple categories: 232  
Total unique shipments: 997  
Percentage with multiple categories: 23.3%

### ! Discussion Questions: Answering A Business Question

**Question 1: Business Question Analysis** - What business question does this comprehensive analysis answer? - How does grouping by ['shipID', 'prodCategory'] differ from grouping by just 'prodCategory'? - What insights can ZappTech's management gain from knowing the percentage of multi-category shipments?

**Briefly Give Answers to the Discussion Questions In This Section**

---

**Question 1: Business Question Analysis**

**i** What business question does this comprehensive analysis answer?

- “How often do shipments span multiple product categories, and what operational or service implications does that have?”

By identifying that **23.3% of shipments contain multiple product categories**, ZappTech gains visibility into the **complexity of order fulfillment**, which directly impacts: - Warehouse picking and packing workflows - Shipment tracking and delay attribution - Customer experience across diverse product lines

**i** How does grouping by ['shipID', 'prodCategory'] differ from grouping by just 'prodCategory'?

| Grouping Type                                    | Purpose   | Usage   |
|--|---|---|
| <code>groupby('prodCategory')</code>             | Aggregates service metrics per <b>category</b> , across all shipments | Understand category-level performance (e.g., Machines vs Liquids)     |
| <code>groupby(['shipID', 'prodCategory'])</code> | Breaks down service metrics <b>per shipment-category pair</b>         | Audit mixed-category shipments, trace delays to specific combinations |

- **Key Difference:** Grouping by ['shipID', 'prodCategory'] exposes intra-shipment variation — crucial when a single shipment includes both fast-moving and slow-moving categories.

**i** What insights can ZappTech’s management gain from knowing the percentage of multi-category shipments?

Knowing that 23.3% of shipments are multi-category enables ZappTech to: 1. **Optimize Fulfillment Operations** - Mixed-category shipments may require **cross-zone coordination** in warehouses. - Helps refine **batching and routing logic** to reduce handling time. 2. **Improve Delay Attribution** - If delays are frequent in multi-category shipments, management can investigate: - Are certain categories consistently causing bottlenecks? - Are packaging or carrier constraints linked to category combinations? 3. **Enhance Customer Communication** - Mixed-category orders may have **split deliveries** or varied service levels. - Enables proactive messaging: “Your furniture will arrive separately from electronics.” 4. **Strategic Product Bundling** - Insights into common category pairings can inform **promotions, bundles, or inventory placement**.

## **i** Visualization of the Service Level by Product Category

```
import matplotlib.pyplot as plt

# Sample data: replace with your actual DataFrame
# Assuming service_by_category is a MultiIndex DataFrame from your groupby/agg
# and 'is_late', 'mean' gives the % late per category
service_level = service_by_category['is_late']['mean'] * 100 # Convert to percentage

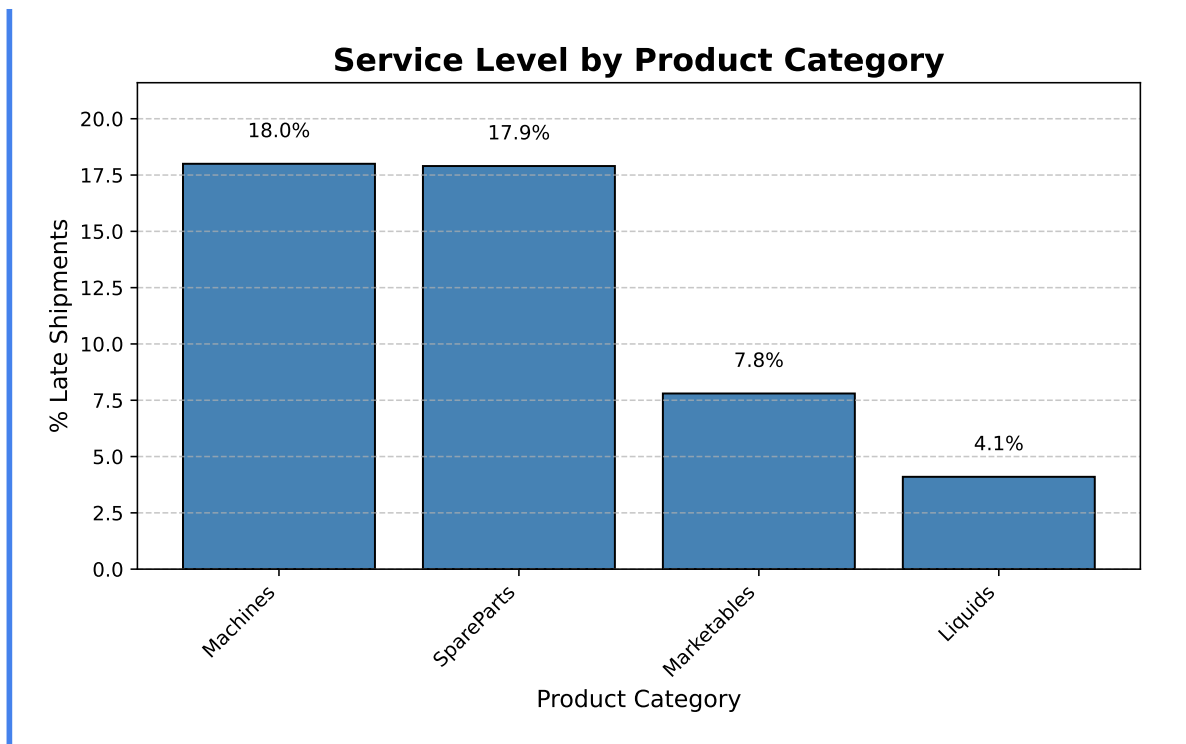
# Sort for better visual impact
service_level = service_level.sort_values(ascending=False)

# Plot
fig, ax = plt.subplots(figsize=(8, 5))
bars = ax.bar(service_level.index, service_level.values, color='steelblue', edgecolor='black')

# Add value labels
for bar in bars:
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, height + 1, f'{height:.1f}%', ha='center', va='bottom')

# Styling
ax.set_title('Service Level by Product Category', fontsize=16, fontweight='bold')
ax.set_ylabel('% Late Shipments', fontsize=12)
ax.set_xlabel('Product Category', fontsize=12)
ax.set_ylim(0, max(service_level.values) * 1.2)
ax.grid(axis='y', linestyle='--', alpha=0.7)

plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



## Student Analysis Section: Mastering Data Manipulation

**Your Task:** Demonstrate your mastery of the seven mental models through comprehensive discussion and analysis. The bulk of your grade comes from thoughtfully answering the discussion questions for each mental model. See below for more details.

### Core Challenge: Discussion Questions Analysis

**For each mental model, provide:** - Clear, concise answers to all discussion questions - Code examples where appropriate to support your explanations

#### ! Discussion Questions Requirements

**Complete all discussion question sections:** 1. **Assign Mental Model:** Data types, date handling, and debugging 2. **Subset Mental Model:** Filtering strategies and complex queries 3. **Drop Mental Model:** Data cleaning and quality management 4. **Sort Mental Model:** Data organization and business logic 5. **Aggregate Mental**

**Model:** Summary statistics and business metrics 6. **Merge Mental Model:** Data integration and quality control 7. **Split-Apply-Combine Mental Model:** Group analysis and advanced operations 8. **Answering A Business Question:** Combining multiple data manipulation techniques to answer a business question

## Professional Visualizations (For 100% Grade)

**Your Task:** Create a professional visualization that supports your analysis and demonstrates your understanding of the data.

**Create visualizations showing:** - Service level (on-time percentage) by product category

**Your visualizations should:** - Use clear labels and professional formatting - Support the insights from your discussion questions - Be appropriate for a business audience - Do not echo the code that creates the visualizations

## Challenge Requirements

**Your Primary Task:** Answer all discussion questions for the seven mental models with thoughtful, well-reasoned responses that demonstrate your understanding of data manipulation concepts.

**Key Requirements:** - Complete discussion questions for each mental model - Demonstrate clear understanding of pandas concepts and data manipulation techniques - Write clear, business-focused analysis that explains your findings

## Getting Started: Repository Setup

### ! Getting Started

**Step 1:** Fork and clone this challenge repository - Go to the course repository and find the “dataManipulationChallenge” folder - Fork it to your GitHub account, or clone it directly - Open the cloned repository in Cursor

**Step 2:** Set up your Python environment - Follow the Python setup instructions above (use your existing venv from Tech Setup Challenge Part 2) - Make sure your virtual environment is activated and the Python interpreter is set

**Step 3:** You’re ready to start! The data loading code is already provided in this file.

**Note:** This challenge uses the same `index.qmd` file you’re reading right now - you’ll edit it to complete your analysis.



## Getting Started Tips

### Method Chaining Philosophy

“Each operation should build naturally on the previous one”

*Think of method chaining like building with LEGO blocks - each piece connects to the next, creating something more complex and useful than the individual pieces.*

### Important: Save Your Work Frequently!

**Before you start:** Make sure to commit your work often using the Source Control panel in Cursor (Ctrl+Shift+G or Cmd+Shift+G). This prevents the AI from overwriting your progress and ensures you don't lose your work.

**Commit after each major step:**

- After completing each mental model section
- After adding your visualizations
- After completing your advanced method chain
- Before asking the AI for help with new code

**How to commit:**

1. Open Source Control panel (Ctrl+Shift+G)
2. Stage your changes (+ button)
3. Write a descriptive commit message
4. Click the checkmark to commit

*Remember: Frequent commits are your safety net!*

## Grading Rubric

**75% Grade:** Complete discussion questions for at least 5 of the 7 mental models with clear, thoughtful responses.

**85% Grade:** Complete discussion questions for all 7 mental models with comprehensive, well-reasoned responses.

**95% Grade:** Complete all discussion questions plus the “Answering A Business Question” section.

**100% Grade:** Complete all discussion questions plus create a professional visualization showing service level by product category.

## Submission Checklist

### Minimum Requirements (Required for Any Points):

- ☐ Created repository named “dataManipulationChallenge” in your GitHub account
- ☐ Cloned repository locally using Cursor (or VS Code)
- ☐ Completed discussion questions for at least 5 of the 7 mental models
- ☐ Document rendered to HTML successfully
- ☐ HTML files uploaded to your repository
- ☐ GitHub Pages enabled and working
- ☐ Site accessible at [https://\[your-username\].github.io/dataManipulationChallenge/](https://[your-username].github.io/dataManipulationChallenge/)

### 75% Grade Requirements:

- ☐ Complete discussion questions for at least 5 of the 7 mental models
- ☐ Clear, thoughtful responses that demonstrate understanding
- ☐ Code examples where appropriate to support explanations

### 85% Grade Requirements:

- ☐ Complete discussion questions for all 7 mental models
- ☐ Comprehensive, well-reasoned responses showing deep understanding
- ☐ Business context for why concepts matter
- ☐ Examples of real-world applications

### 95% Grade Requirements:

- ☐ Complete discussion questions for all 7 mental models
- ☐ Complete the “Answering A Business Question” discussion questions
- ☐ Comprehensive, well-reasoned responses showing deep understanding
- ☐ Business context for why concepts matter

### 100% Grade Requirements:

- ☐ All discussion questions completed with professional quality
- ☐ Professional visualization showing service level by product category
- ☐ Professional presentation style appropriate for business audience
- ☐ Clear, engaging narrative that tells a compelling story
- ☐ Practical insights that would help ZappTech’s management

### Report Quality (Critical for Higher Grades):

- ☐ Professional writing style (no AI-generated fluff)
- ☐ Concise analysis that gets to the point