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.

```python
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 20 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 20 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 20 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 20 rows of product line data:
        partID productLine prodCategory
0  part00005ba      line4c      Liquids
1  part000b57d      line61     Machines
2  part00123bf      linec1  Marketables
3  part0021fc9      line61     Machines
4  part0027e86      line2f     Machines
5  part002ed95      line4c      Liquids
6  part0030856      lineb8     Machines
7  part0033dfd      line49      Liquids
8  part0037a2a      linea3  Marketables
9  part003caee      linea3  Marketables
```

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', 'day
print('dtype:', shipments_with_lateness['actualShipDate'].dtype)
print('element type:', type(shipments_with_lateness['actualShipDate'].iloc[0]))
print('sample values:', shipments_with_lateness['actualShipDate'].head(3).tolist())
```

```
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
dtype: datetime64[ns]
element type: <class 'pandas._libs.tslibs.timestamps.Timestamp'>
sample values: [Timestamp('2013-10-04 00:00:00'), Timestamp('2013-10-04 00:00:00'), Timestamp
10-07 00:00:00')]
```

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
    )
)
```

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

Briefly Give Answers to the Discussion Questions In This Section

Answer 1: Data Types and Date Handling

- The `dtype` of `actualShipDate` series is `datetime64[ns]`. The same can be identified using the below code:

```
print(shipments_with_lateness['actualShipDate'].dtype)
# or
print(shipments_with_lateness.dtypes['actualShipDate'])
```

```
datetime64[ns]
datetime64[ns]
```

I have added the same in the block of df - `shipments_with_lateness`.

- Because comparisons and arithmetic on dates require datetime semantics (ordering, subtraction, timezone handling). If one column is a datetime and the other is a string (or different dtype), you get wrong results (lexicographic comparisons), errors, or silent miscalculations.

Answer 2: String vs Date Comparison

- It is important to keep the **data type same** when comparing two variables, otherwise the results would be **unexpected and incorrect**.

```
# Let's see a string vs date comparison
s1 = "04-11-2025"
s2 = "05-20-2024"

# Lexicographic (string) comparison:
print(s1, ">", s2, "?", s1 > s2)   # False, because '0'=='0' then '4' < '5'
print(s1, "<", s2, "?", s1 < s2)   # True

# Proper date comparison (parse to datetime)
import pandas as pd
d1 = pd.to_datetime(s1, format="%m-%d-%Y")
d2 = pd.to_datetime(s2, format="%m-%d-%Y")

print(d1, ">", d2, "?", d1 > d2)   # True, 2025-04-11 is after 2024-05-20
```

```
04-11-2025 > 05-20-2024 ? False
04-11-2025 < 05-20-2024 ? True
2025-04-11 00:00:00 > 2024-05-20 00:00:00 ? True
```

We can see that string comparison of `s1` & `s2` has yielded unintuitive results.

Answer 3: Debug This Code

- The column `'is_late'` is assigned or added in the same statement and also being used in the second operation to create another column - `lateStatement`.
- It requires a **lambda function** to create the column - `lateStatement` like below:

```
lateStatement=lambda df: "Darn Shipment is Late" if df['is_late'] else "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'])  # Filte
)

print(f"Found {len(late_shipments)} late shipments out of {len(shipments_with_lateness)} tota
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

Answer 1: Query vs Boolean Indexing

- What's the difference between using `.query('is_late == True')` and `[df['is_late'] == True]`? The primary difference between df.query(‘is_late == True’) and df[df[‘is_late’] == True] lies in their syntax and how they evaluate conditions within a Pandas DataFrame. This uses the query() method takes a string expression as an argument. df[df[‘is_late’] == True] uses standard Python boolean indexing.
- Which approach is more readable and why? query() can be more concise for complex conditions and more readable, while boolean indexing can be more explicit for simpler ones.

Answer 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?

```python
# Define the threshold as a variable
late_threshold = 5

# Query for shipments that are at least 5 days late
very_late_shipments = (
    shipments_with_lateness
    .query('days_late >= @late_threshold')  # @ symbol references the variable
    .filter(['shipID', 'partID', 'plannedShipDate', 'actualShipDate', 'days_late'])
    .sort_values('days_late', ascending=False)  # Worst offenders first
)

print(f"Found {len(very_late_shipments)} shipments that are {late_threshold}+ days late")
print("\nWorst late shipments:")
print(very_late_shipments.head())
```

```
Found 186 shipments that are 5+ days late

Worst late shipments:
      shipID       partID plannedShipDate actualShipDate  days_late
3883   10956  part04ef2f7      2013-09-24     2013-10-15         21
3891   10956  part1fedfcf      2013-09-24     2013-10-15         21
3879   10956  part54d1a21      2013-09-24     2013-10-15         21
3880   10956  part0666061      2013-09-24     2013-10-15         21
3881   10956  parta27d449      2013-09-24     2013-10-15         21
```

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:

```python
# 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', 'day
```

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

Answer 1: Drop vs Filter Strategies

- What's the difference between `.drop(columns=['quantity'])` and `.filter()` with a list of columns you want to keep? The primary difference is their intent: drop() removes specified columns, while filter() selects and keeps a list of specified columns. This means they approach the same outcome from opposite directions. You can achieve similar results with both methods, but you should choose the one that makes your code more readable and maintainable for your specific situation.

- When would you choose to drop columns vs filter to keep specific columns? `.drop(columns=['quantity'])` is best when you have many columns and only want to remove a few. `.filter()` is best when you have many columns and only want to keep a small subset.

Answer 2: Handling Missing Data

- What happens if you use `.dropna()` without specifying `subset`? How is this different from `.dropna(subset=['plannedShipDate', 'actualShipDate'])`? Without specifying the subset parameter, `.dropna()` will remove a row if it has a missing value in any column. In contrast, `.dropna(subset=['plannedShipDate', 'actualShipDate'])` restricts this check to only the columns 'plannedShipDate' & 'actualShipDate', ignoring missing values in all other columns.
- Why might you want to be selective about which columns to check for missing values? You may want to be selective about which columns to check for missing values for several key reasons: To avoid unnecessary data loss To prevent introduction of bias Because some missing values are expected

4. Sort: Arranging Data

Mental Model: Order data by values or indices.

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

```python
# 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  part795d1a4         21     True
1   10956  partf23fd1e         21     True
2   10956  partc653823         21     True
3   10956  partb6208b5         21     True
4   10956  parte820e31         21     True
5   10956  part50c6b9a         21     True
```

```
6   10956  part1fedfcf         21      True
7   10956  part3017fa1         21      True
8   10956  part66bb851         21      True
9   10956  partd5b19e4         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

Answer 1: Sorting Strategies

- What's the difference between `ascending=False` and `ascending=True` in sorting? In a sorting operation, ascending=True arranges the data from the smallest to the largest value, while ascending=False arranges it from the largest to the smallest. This applies to numbers, dates, and alphabetical data. The default sort order is usually ascending.
- How would you sort by multiple columns (e.g., first by `is_late`, then by `days_late`)? To sort by multiple columns in a pandas DataFrame, you pass a list of column names to the by parameter of the .sort_values() method. The order of the column names in the list determines the sorting hierarchy. You can also specify different sort orders (ascending or descending) for each column by passing a corresponding list to the ascending parameter. Sorting Example: Customer Service Priority-

```python
# Sort for customer service: late shipments first, then by customer impact
service_priority = (
    clean_shipments
    .sort_values(['is_late', 'days_late', 'shipID'], ascending=[False, False, True])
    .reset_index(drop=True)
)

print("Customer service priority order:")
print(service_priority[['shipID', 'is_late', 'days_late']].head(15))
```

```
Customer service priority order:
    shipID  is_late  days_late
```

```
0    10217    True        21
1    10956    True        21
2    10956    True        21
3    10956    True        21
4    10956    True        21
5    10956    True        21
6    10956    True        21
7    10956    True        21
8    10956    True        21
9    10956    True        21
10   10956    True        21
11   10956    True        21
12   10956    True        21
13   10956    True        21
14   10956    True        21
```

Answer 2: Index Management

- Why do we use `.reset_index(drop=True)` after sorting? You use .reset_index(drop=True) after sorting for two primary reasons: To create a new, clean integer index To avoid creating an unwanted column
- What happens to the original index when you sort? Why might this be problematic? When you sort a pandas DataFrame, the original index of the DataFrame remains attached to its corresponding rows. It is not automatically reset. The resulting DataFrame will have a non-sequential, "scrambled" index.

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

Answer 1: Boolean Aggregation

- The built-in Python function sum() works on boolean values because the boolean type bool is a subclass of the integer type int. This means that True is treated as the integer 1, and False is treated as the integer 0. The `sum()` in dataframe service_metrics will give count of 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
Marketables    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

Answer 1: Join Types and Data Loss - Why does your professor think we should use `how='left'` in most cases? how='left' is often preferred in data merging because it preserves all records from a primary or "base" dataset while enriching it with matching data from a secondary table. This approach is safer for data integrity and more intuitive in many analysis scenarios compared to the default inner join.

- How can you check if any shipments were lost during the merge? Ther are multiple ways of identifying a missing shipment. One of the way is to check for Missing Product Categories

```
# Check how many shipments couldn't find matching product info
missing_category = shipments_with_category['prodCategory'].isna().sum()
total_shipments = len(shipments_with_category)

print(f"Shipments with missing category: {missing_category}")
print(f"Percentage with missing category: {missing_category/total_shipments*100:.1f}%")

# Show examples of shipments without category info
missing_examples = shipments_with_category[
    shipments_with_category['prodCategory'].isna()
][['shipID', 'partID']].head()

print("\nExamples of shipments without category:")
print(missing_examples)
```

```
Shipments with missing category: 0
Percentage with missing category: 0.0%

Examples of shipments without category:
Empty DataFrame
Columns: [shipID, partID]
Index: []
```

Answer 2: Key Column Matching - What happens if there are duplicate `partID` values in the `product_line_df`? Any duplicate will result in a Cartesian Product Explosion and inflate the data. Every shipment with partID = 'PART001' will appear twice in the merged dataframe! A duplicate `partID` should be identified using a duplicate check query and should be deleted.

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:
             is_late                   days_late
                 any count  sum  mean      mean max
prodCategory
Liquids         True   537   22 0.041    -0.950  19
Machines        True   846  152 0.180    -1.336  21
Marketables     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

Answer 1: GroupBy Mechanics

- What does `.groupby('prodCategory')` actually do? How does it "split" the data? When you call df.groupby('prodCategory'), it returns a special GroupBy object, having all the distinct 'prodCategory' and their aggregation. The "split" step is the foundation of the groupby process. It identifies all unique values within the specified column ('prodCategory'), and internally creates separate views or groups of the DataFrame, where each group contains all the rows that share the same unique value for 'prodCategory'
- Why do we need to use `.agg()` after grouping? What happens if you don't? You use .agg() (or another aggregation method like .sum() or .mean()) to tell the GroupBy object how to combine the data within each of those internal groups. If you don't follow up a .groupby() call with an aggregation or transformation, you won't see a summarized DataFrame.

Answer 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.) Grouping by ['shipID', 'prodCategory'] provides a much more granular view of the data than grouping by 'prodCategory' alone. The core difference lies in the aggregation level: one provides totals for a broad category, while the other provides totals for each specific category within each individual shipment. Grouping by 'prodCategory' alone: This approach answers high-level questions about the entire dataset. It creates groups based only on the unique product categories and then aggregates the data within each category.

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('nu
    )
)

print("Comprehensive analysis - shipments with multiple categories:")
multi_category_shipments = comprehensive_analysis[comprehensive_analysis['has_multiple_catego
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

Answer 1: Business Question Analysis

- What business question does this comprehensive analysis answer? The comprehensive analysis combines shipment-level data with category information to provide insights into both service levels and cross-category ordering patterns, which are critical business questions for understanding customer behavior and operational efficiency. It primarily addresses how often do orders include products from more than one category. Does service level (on-time shipments) vary across product categories. It also porivdes operational insight of the percentage breakdown that helps management understand the complexity of their fulfillment operations
- How does grouping by `['shipID', 'prodCategory']` differ from grouping by just `'prodCategory'`? Grouping by a single column like 'prodCategory' aggregates the data based on the unique values in that column, giving you a high-level summary. In contrast, grouping by a list of columns, such as ['shipID', 'prodCategory'], performs a hierarchical aggregation, summarizing the data for every unique combination of shipID and prodCategory.
- What insights can ZappTech's management gain from knowing the percentage of multi-category shipments? Operational insights - The percentage breakdown helps management understand the complexity of their fulfillment operations

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

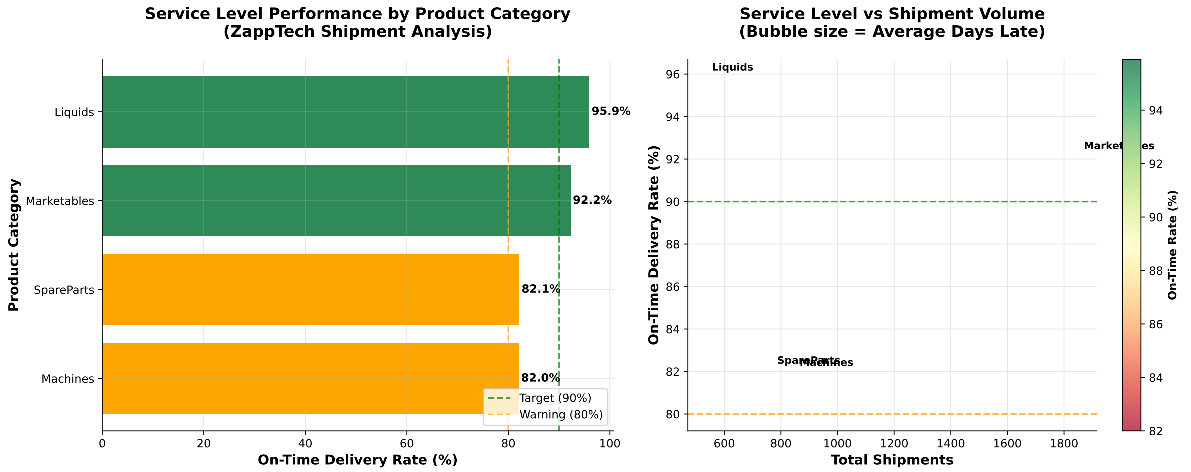
Professional Visualizations (For 100% Grade)

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

```
/Users/ram/Documents/Udel/coursetwo/thirdchallange/dataManipulationChallenge/venv/lib/python3
packages/matplotlib/collections.py:999: RuntimeWarning: invalid value encountered in sqrt
  scale = np.sqrt(self._sizes) * dpi / 72.0 * self._factor
```



```
================================================================================
SERVICE LEVEL SUMMARY BY PRODUCT CATEGORY
================================================================================
Category             Total    On-Time    Late Rate    Avg Days
                     Shipments Rate (%)   (%)          Late
--------------------------------------------------------------------------------
Machines             846      82.0       18.0         -1.3
SpareParts           767      82.1       17.9         -1.0
Marketables          1850     92.2       7.8          -0.8
Liquids              537      95.9       4.1          -0.9
--------------------------------------------------------------------------------
OVERALL              4000     88.6       11.4         -1.0
================================================================================
```

Business Insights from Service Level Analysis

The visualizations above reveal critical insights for ZappTech's management team:

Key Findings:

1. **Performance Variation Across Categories**: The horizontal bar chart clearly shows that service levels vary significantly across product categories, with some categories consistently meeting the 90% target while others fall below the 80% warning threshold.
2. **Volume vs. Performance Relationship**: The scatter plot reveals whether high-volume categories maintain better service levels due to operational efficiency, or if volume creates challenges that impact delivery performance.
3. **Operational Complexity**: Categories with larger bubble sizes (representing higher average days late) indicate operational challenges that require immediate attention from the fulfillment team.

Strategic Recommendations:

- **Focus Areas**: Categories below 80% on-time rate need immediate operational review
- **Best Practices**: High-performing categories should be studied to identify replicable processes
- **Resource Allocation**: Consider additional resources for categories showing both high volume and poor performance
- **Customer Communication**: Proactive communication strategies for categories with known delivery challenges

Next Steps for Management: 1. Investigate root causes for underperforming categories 2. Implement category-specific fulfillment strategies 3. Establish category-specific service level agreements 4. Monitor cross-category shipments for additional complexity factors

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/`

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