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 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  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:

```python
# 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', 'da
```

```
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
    )
)
```

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

Briefly Give Answers to the Discussion Questions In This Section

Answer to Question 1:

- The `dtype` of the `actualShipDate` series is `datetime64[ns]`
- We can find out using various methods as follows:

```
# Method 1: Using .dtype attribute
print("actualShipDate dtype:", shipments_df['actualShipDate'].dtype)
print("================================================")

# Method 2: Using .dtypes (for the entire dataframe)
print("All column dtypes:")
print(shipments_df.dtypes)
print("================================================")
# Method 3: More detailed info with .info()
print("Detailed dataframe info:")
shipments_df.info()
print("================================================")
```

```
actualShipDate dtype: datetime64[ns]
================================================
All column dtypes:
shipID                      int64
plannedShipDate    datetime64[ns]
actualShipDate     datetime64[ns]
partID                     object
```

```
quantity                    int64
dtype: object
================================================
Detailed dataframe info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4000 entries, 0 to 3999
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   shipID           4000 non-null   int64
 1   plannedShipDate  4000 non-null   datetime64[ns]
 2   actualShipDate   4000 non-null   datetime64[ns]
 3   partID           4000 non-null   object
 4   quantity         4000 non-null   int64
dtypes: datetime64[ns](2), int64(2), object(1)
memory usage: 156.4+ KB
================================================
```

- It is important that both `actualShipDate` and `plannedShipDate` have the same data type for comparison operations to work correctly. Pandas will compare the dates as strings if they are not the same type and comparisons would be alphabetical rather than chronological.

Answer to Question 2:

When treated as strings, the comparison happens character by character.

```
- "04-11-2025" is compared to "05-20-2024".
- The first character 0 is the same.
- The second character 4 is compared to 5.
- Because 4 comes before 5 alphabetically, "04-11-2025" is considered "less than" "05-
20-2024", which is wrong chronologically
```

```python
from datetime import datetime
# The dates as strings
date_string_1 = "04-11-2025"
date_string_2 = "05-20-2024"

print(f"Comparing dates as strings:")
print(f"'{date_string_1}' > '{date_string_2}' is {date_string_1 > date_string_2}")
print(f"'{date_string_1}' < '{date_string_2}' is {date_string_1 < date_string_2}\n")

# The format of our dates is MM-DD-YYYY
```

```python
date_format = "%m-%d-%Y"

# Convert the strings to datetime objects
date_object_1 = datetime.strptime(date_string_1, date_format)
date_object_2 = datetime.strptime(date_string_2, date_format)

print(f"Comparing dates as datetime objects:")
print(f"'{date_string_1}' > '{date_string_2}' is {date_object_1 > date_object_2}")
print(f"'{date_string_1}' < '{date_string_2}' is {date_object_1 < date_object_2}\n")
```

```
Comparing dates as strings:
'04-11-2025' > '05-20-2024' is False
'04-11-2025' < '05-20-2024' is True

Comparing dates as datetime objects:
'04-11-2025' > '05-20-2024' is True
'04-11-2025' < '05-20-2024' is False
```

Answer to Question 3:

- lamdba-df was missing to reference the dataframe
- lateStatement is a string, not a boolean so we need use vectorized operations to create the string like np.where()

```python
#This is the corrected code
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 
    )
)
```

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:

```python
# 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 to Question 1:

- In pandas, df[df['is_late'] == True] uses boolean indexing, while df.query('is_late == True') uses the .query() method. While they can produce the same result for simple cases, they operate differently and have trade-offs in terms of readability, performance, and flexibility.
- The .query() method is more readable and allows for more complex queries using SQL-like syntax.

Answer to Question 2:

```
#Set the late threshold
late_threshold = 5

#Query the shipments that are at least 5 days late using the late_threshold variable
late_shipments = (
   shipments_with_lateness
   .query('days_late >= @late_threshold')
   .filter(['shipID', 'partID', 'plannedShipDate', 'actualShipDate', 'days_late'])
)
print(f"Found {len(late_shipments)} shipments that are at least {late_threshold} days late ou
```

```
Found 186 shipments that are at least 5 days late out of 4000 total
```

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, 7 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

Answer to Question 1:

- `.drop(columns=['quantity'])` removes that specific column from the dataframe. You would choose to drop columns if you don't need them for your analysis.
- `.filter()` explicitly specifies the columns you want to retain. You would create a new DataFrame by selecting only the columns present in your list.

Answer to Question 2:

- If you use `.dropna()` without specifying `subset`, it will remove all rows with any missing values in any column.
- This is different from `.dropna(subset=['plannedShipDate', 'actualShipDate'])` which will only remove rows with missing values in the `plannedShipDate` and `actualShipDate` columns.
- You might want to be selective about which columns to check for missing values if you have a lot of columns and you only want to check a few of them.

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
)
```

```python
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  partb6208b5         21     True
1   10956  part04ef2f7         21     True
2   10956  part4875f85         21     True
3   10956  partb722d53         21     True
4   10956  partc979912         21     True
5   10956  parta27d449         21     True
6   10956  partc653823         21     True
7   10956  part82e69e9         21     True
8   10956  partf23fd1e         21     True
9   10956  part825873c         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 to Question 1:

- ascending=True:
  - This sorts data in ascending order.
  - For numerical values, it means from smallest to largest (e.g., 1, 2, 3).
  - For alphabetical values, it means from A to Z (e.g., Apple, Banana, Cherry).
  - For dates, it means from earliest to latest (e.g., 2023-01-01, 2023-01-02).
- ascending=False:
  - This sorts data in descending order.
  - For numerical values, it means from largest to smallest (e.g., 3, 2, 1).
  - For alphabetical values, it means from Z to A (e.g., Cherry, Banana, Apple).
  - For dates, it means from latest to earliest (e.g., 2023-01-02, 2023-01-01).

- To sort by multiple columns, such as first by is_late and then by days_late, you typically provide a list of column names to the sorting function or method. The order of columns in this list determines the priority of sorting. The first column in the list is the primary sort key, and subsequent columns are used to break ties within the primary sort.

```
# Sort first by is_late (False first, then True), then by days_late
sorted_shipments = (
    clean_shipments
    .sort_values(
        by=['is_late', 'days_late'],
        ascending=[True, False]  # is_late ascending (False first), days_late descending
    )
    .reset_index(drop=True)
)


print("Shipments sorted by first is_late and then by days_late :")
print(sorted_shipments[['shipID', 'partID', 'days_late', 'is_late']].head(50))
```

```
Shipments sorted by first is_late and then by days_late :
    shipID       partID  days_late  is_late
0    10193  part71b107e          0    False
1    10193  part8acdaf4          0    False
2    10193  part3abf99d          0    False
3    10193  part6ee349a          0    False
4    10193  part7259bf2          0    False
5    10193  partfe2de18          0    False
6    10193  part82548d0          0    False
7    10193  partb60c5be          0    False
8    10193  partf194391          0    False
9    10193  partb8509b1          0    False
10   10193  parte04a6ce          0    False
11   10193  partfcb753c          0    False
12   10193  part0e8d20e          0    False
13   10193  part2302c27          0    False
14   10193  partd578ab3          0    False
15   10193  part15541ff          0    False
16   10193  part6527169          0    False
17   10193  part9146d41          0    False
18   10193  part57669fb          0    False
19   10193  part21bade2          0    False
20   10193  partba76e17          0    False
```

```
21   10193  partef9a92a         0    False
22   10193  parta2cc65f         0    False
23   10193  part481932d         0    False
24   10193  partcbde9e0         0    False
25   10193  part472d67c         0    False
26   10193  part7547885         0    False
27   10193  parta442b41         0    False
28   10193  partf557daa         0    False
29   10207  part0e9dcf0         0    False
30   10207  part283dfaa         0    False
31   10207  partcfdcff7         0    False
32   10207  part5f49e4d         0    False
33   10207  part0c5bf9a         0    False
34   10207  part5107513         0    False
35   10207   part05257e         0    False
36   10207  part07bee5e         0    False
37   10207  part08d5f32         0    False
38   10207   parte310d1         0    False
39   10207  part2e3c85b         0    False
40   10207  part8fb9837         0    False
41   10207  part8100a77         0    False
42   10207   part74dc9f         0    False
43   10207  part198d6c2         0    False
44   10207  part1d3bfad         0    False
45   10207  partfd372fb         0    False
46   10207  part1d8d910         0    False
47   10207  partc1987d8         0    False
48   10207  part07671a5         0    False
49   10207  parteea7762         0    False
```

First sort by is_late:

- False (on-time) shipments come first
- True (late) shipments come second

Then sort by days_late within each group:

- For on-time shipments: most negative (earliest) first
- For late shipments: most positive (latest) first

Answer to Question 2:

- Using .reset_index(drop=True) after sorting a Pandas DataFrame serves to re-establish a default, sequential integer index while discarding the old, potentially non-sequential or non-numeric index.
- Maintaining a Clean, Sequential Index: When a DataFrame is sorted, the original index labels remain associated with their respective rows, even though the order of the rows changes. This can lead to a non-sequential or jumbled index. reset_index() replaces this with a new, clean index starting from 0.
- Facilitating Positional Indexing: A sequential integer index makes it easier to access rows by their position using iloc, as the index directly corresponds to the row number.
- Preventing Misleading Index Values: If the original index held no inherent meaning beyond identifying the row's original position, retaining it after sorting can be misleading, as the index values no longer reflect the current order of the data.
- drop=True's Role: The drop=True argument is crucial here because it prevents the old index from being added as a new column in the DataFrame. If drop=True were omitted, the old index would be converted into a regular column, which is often not the desired behavior after sorting.

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 percentag
        '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 to Q1:

- In Pandas, the sum() works on boolean values because bool is a subclass of int in Python. This means that True is treated as 1 and False is treated as 0 when numerical operations are performed.
- When we call sum() on a Series or DataFrame column containing boolean values, it effectively counts the number of True values present. Each True value contributes 1 to the sum, while each False value contributes 0.

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:

```python
# 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 to Q1:

- Using how='left' in .merge() is often preferred because it guarantees that all rows from the "left" DataFrame are preserved in the merged result. how='left' is a common and safe default when the goal is to add information to an existing, primary dataset without losing any of its original records. This is crucial in many data analysis scenarios for several reasons:
  - Data Integrity: It ensures that no data from your primary (left) DataFrame is accidentally dropped due to a lack of matching keys in the "right" DataFrame. This helps maintain the integrity of your original dataset.
  - Adding Contextual Information: When you want to enrich an existing dataset with additional information from another source, a left merge allows you to add those details without losing any of your original observations. If a match isn't found in the right DataFrame, the corresponding columns will simply be filled with NaN values, indicating the absence of that specific information for those rows.
  - Predictability and Control: Left merges offer more predictable behavior regarding the number of rows in the output. The resulting DataFrame will always contain at least as many rows as the left DataFrame.
  - Debugging and Traceability: If you need to investigate why certain rows lack information after a merge, the NaN values resulting from a left merge clearly indicate which rows did not find a match in the right DataFrame. This makes debugging and understanding your data easier.

To check if any shipment were lost during the merge, we can verify the using various ways such as :

```
- Compare before / after row counts.- Check for any rows with missing values.
- Check for any rows having duplicate values.
```

Here is an example code for Comparing row counts:

```
# Check row counts before and after merge
print(f"Original shipments: {len(clean_shipments)}")
print(f"After merge: {len(shipments_with_category)}")
print(f"Difference: {len(shipments_with_category) - len(clean_shipments)}")

# Verify no data loss
if len(shipments_with_category) == len(clean_shipments):
    print("No shipments lost during merge")
else:
    print("Data loss detected!")
```

```
Original shipments: 4000
After merge: 4000
Difference: 0
No shipments lost during merge
```

Answer to Q2:

If there are duplicate `partID` values in the `product_line_df`, we can run into data integrity issues. In his scenario, merge will create multple rows for each shipment that matches those duplicate partIDs.

Here is an example:

- product_line_df has :

  | partID | productLine | productCategory

  | part123 | Line1 | Machines

  | part123 | Line2 | Liquids #Duplicate partID

- While merging with shipments_df:

  Each shipment with partID "part123" will get BOTH rows. As a Result, 1 shipment becomes 2 rows in the merged data.

7. Split-Apply-Combine: Group Analysis

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

Now let's analyze service levels by category:

```python
# 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 to Q1:

- In pandas, groupby combines "split-apply-combine" process into one with the help of .agg().
- Just `.groupby('prodCategory')` will create a GroupBy object that contains instructions for how to split the data based on a 'productCategory'. The groupby operation then creates separate data chunks for each unique value in grouping column `productCategory`.
- You need .agg() after grouping to perform specific calculations (like sum, mean, max) on each group, reducing the data to a summarized, single row per group. It allows you to apply one or more aggregation functions to a DataFrame, making it much easier to summarize and analyze data. Without an aggregation function, the grouping operation itself would not produce a useful, summarized result.
- If you group a DataFrame but do not specify an aggregation, you are left with a "GroupBy object," which is not the final result you typically want. This object is an intermediate step in the "split-apply-combine" process, and you must apply an aggregation function (like .agg(), .sum(), or .mean()) to the "apply" and "combine" those results into a meaningful output.

Answer to Q2:

```
Grouping by 'prodCategory' alone:

    - Question: "What is the overall service level for each product category?"
    - Answer: Service metrics aggregated across ALL shipments for each category
    - How many total shipments in each category?
    - What percentage of shipments are late in each category?
    - Average and maximum lateness by category

Grouping by ['shipID', 'prodCategory']:
    - Question: "What is the service level for each shipment within each product category?"
    - Answer: Service metrics for each individual shipment within each category
    - Each row represents one shipment's performance in one category
    - Shows which specific shipments had issues in which 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('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 to Q1:

The Business Question This Comprehensive Analysis Answers Primary Question:

- "How often do our customers place orders that span multiple product categories, and what are the operational implications of these cross-category orders?"

- Grouping by `['shipID', 'prodCategory']` creates one row per shipmentID-prodcutCategory combination as compared to just producCatgory which would have created one row each per category only.
- It then idenifies shipments that contain products from more than one category.
- Finally, shows what amount of all shipments are multi=category as percentage.

Business insights ZappTech's management can know from percentage of multi-category shipments:

- Cross-Category Shopping behavior: 232 out of 997 (23.3%) contain multiple productCategories. This tells about customer ordering pattrns.

  Business Question: "Do our customers typically order from one category at a time, or do they frequently mix categories in single orders?"

- Operational Complexity

  Business Question: "What percentage of our shipments require coordination across multiple product lines, and how does this affect our fulfillment process?"

- Service Level Impact

  Business Question: "Are multi-category orders more likely to experience delays due to the complexity of coordinating different product lines?"

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/`

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