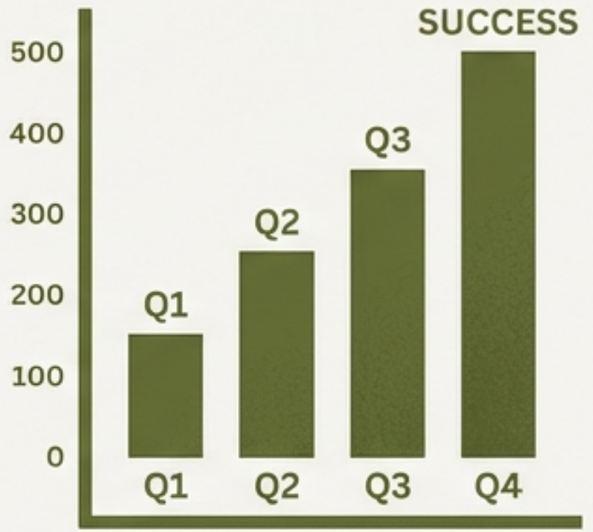
The PyCafe Case Study: Turning Raw Receipts into Business Strategy with Pandas

A methodology for transforming messy operational data into actionable business intelligence.



Key Topics Covered: Data Structuring, Inspection, Cleaning, Manipulation, Analysis, and Reporting.

PyCafe's Challenge: Critical Business Questions Blocked by Messy Data

You are the manager of PyCafe, a new coffee shop. Your digital receipts from the first few days are your only source of truth, but they are unreliable due to data entry errors.

THE DATA

OrderID	Date	Item	Price
OrderID: 1003	2023-10-27	Latte	4.50
OrderID: 1003	2023-10-27	Latte	4.50
OrderID: 1005	2023-10-27	Muffin	NaN

THE PROBLEM

- Duplicate transaction entries are inflating revenue figures.
- Missing price data makes profit calculation impossible.
- Inconsistent formatting prevents trend analysis.

THE GOAL: CRITICAL QUESTIONS

- Are we profitable?
- Who is our top-performing barista?
- What are the daily sales trends?

Our Methodology: A 7-Phase Journey from Raw Data to Report

We will follow a structured, repeatable process to transform PyCafe's raw data. Each phase builds upon the last, leading us from chaos to clarity.

Setup

Structuring the raw data.

Inspect

Diagnosing the issues.

Clean

Establishing a single source of truth.

Enrich

Engineering new features for analysis.

Query

Asking targeted questions.

Analyze

Aggregating data to find patterns.

Report

Synthesizing the final insights.

Phase 1: Organizing Receipts into a Structured DataFrame

Before any analysis can begin, we must load the data from individual receipts into a single, table-like structure that is programmable.

The How - Visual Concept

```
{'Item':
  ['Latte',
   'Espresso'],
 'Price':
  [4.50,
  3.00]}
```

Item	Price
Latte	4.50
Espresso	3.00

The How - Practice

Code Snippet 1 (Conceptual)

```
# Concept: A DataFrame is like a programmable
spreadsheet
data = {'Item': ['Latte', 'Espresso'], 'Price':
[4.50, 3.00]}
df = pd.DataFrame(data)
```

Code Snippet 2 (Real-World)

```
# Practice: In the real world, we load data from a
file
df = pd.read_csv('pycafe_receipts.csv')
```

Phase 2: Diagnosing the Data's Health and Integrity

Calculating metrics on flawed data leads to incorrect business conclusions. We must first identify all structural issues, missing values, and incorrect data types.

Key Tool: `.head()` - Peek at the data

OrderID	Date	Item	Price	Quantity
101	2023-01-15	Latte	4.50	1
102	2023-01-15	Espresso	3.00	2
103	2023-01-16	Muffin	NaN	1
103	2023-01-16	Muffin	2.75	1
104	2023-01-17	Coffee	2.50	1

```
df.head()
```

Key Tool: `.info()` - Get a technical summary

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9 entries, 0 to 8
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   OrderID   9 non-null      int64
 1   Date      9 non-null      object
 2   Item      9 non-null      object
 3   Price     8 non-null      float64
 4   Quantity  9 non-null      int64
dtypes: float64(1), int64(2), object(2)
memory usage: 488.0 bytes
```

Price has missing values (8 out of 9 entries are not null).

'Date' is an 'object' (text) type, not a proper date. This will prevent time-based analysis.

Phase 3: Establishing a Single Source of Truth by Cleaning Data

To ensure accuracy in our final report, we must correct errors by removing duplicate transactions and intelligently handling missing information.

Before & After

Problem 1: Duplicate Transactions

Before

OrderID	Date	Item	Price	Quantity
101	2023-01-15	Latte	4.50	1
102	2023-01-15	Espresso	3.00	2
103	2023-01-16	Muffin	2.75	1
103	2023-01-16	Muffin	2.75	1

```
df.drop_duplicates(subset=['OrderID'])
```

Crucial. It targets specific transaction errors, not legitimate identical orders (e.g., two different customers ordering an Espresso).

After

OrderID	Date	Item	Price	Quantity
101	2023-01-15	Latte	4.50	1
102	2023-01-15	Espresso	3.00	2
103	2023-01-16	Muffin	2.75	1

Problem 2: Missing Price Data

Before

OrderID	Date	Item	Price	Quantity
101	2023-01-15	Latte	4.50	1
102	2023-01-15	Espresso	3.00	2
104	2023-01-17	Coffee	2.50	1
105	2023-01-17	Muffin	NaN	1

```
avg_price = df['Price'].mean()
df['Price'].fillna(value=avg_price, inplace=True)
```

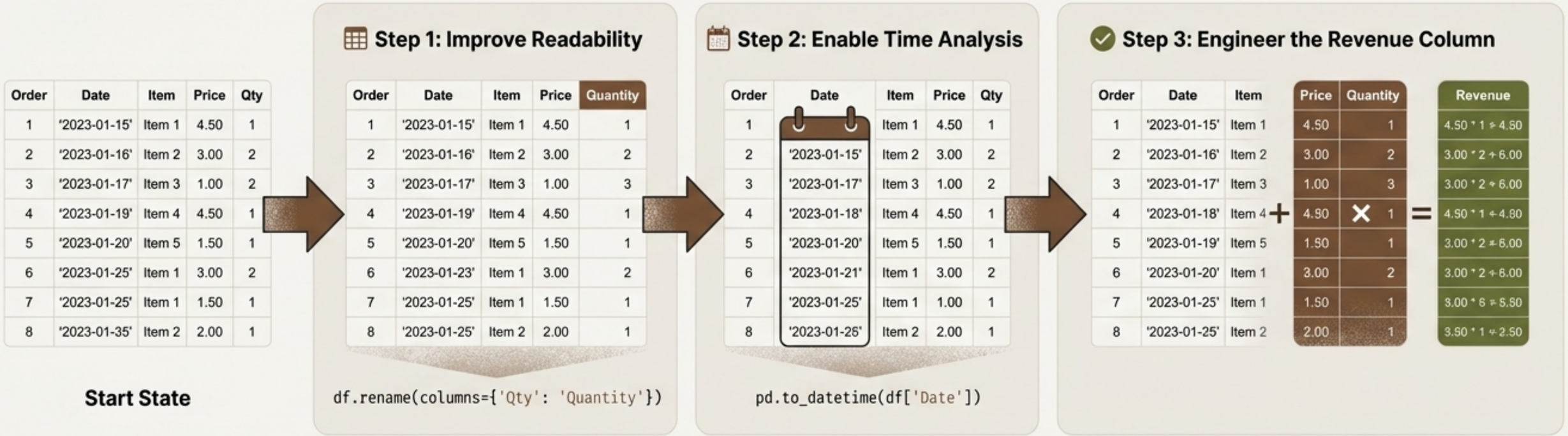
Strategy: We are using imputation—filling the missing Muffin price with the average price of other items—as a reasonable estimate to avoid losing the entire sales record.

After

OrderID	Date	Item	Price	Quantity
101	2023-01-15	Latte	4.50	1
102	2023-01-15	Espresso	3.00	2
104	2023-01-17	Coffee	2.50	1
105	2023-01-17	Muffin	3.33	1

Phase 4: Enriching the Data for Deeper Analysis

The raw data tells us *what* was sold, but not its *value*. We need to engineer new columns, like 'Revenue', and correct data types to enable financial calculations and time-series analysis.



Order	Date	Item	Price	Qty
1	'2023-01-15'	Item 1	4.50	1
2	'2023-01-16'	Item 2	3.00	2
3	'2023-01-17'	Item 3	1.00	2
4	'2023-01-19'	Item 4	4.50	1
5	'2023-01-20'	Item 5	1.50	1
6	'2023-01-25'	Item 1	3.00	2
7	'2023-01-25'	Item 1	1.50	1
8	'2023-01-35'	Item 2	2.00	1

Order	Date	Item	Price	Quantity
1	'2023-01-15'	Item 1	4.50	1
2	'2023-01-16'	Item 2	3.00	2
3	'2023-01-17'	Item 3	1.00	3
4	'2023-01-19'	Item 4	4.50	1
5	'2023-01-20'	Item 5	1.50	1
6	'2023-01-23'	Item 1	3.00	2
7	'2023-01-25'	Item 1	1.50	1
8	'2023-01-25'	Item 2	2.00	1

Order	Date	Item	Price	Qty
1	'2023-01-15'	Item 1	4.50	1
2	'2023-01-16'	Item 2	3.00	2
3	'2023-01-17'	Item 3	1.00	2
4	'2023-01-18'	Item 4	4.50	1
5	'2023-01-20'	Item 5	1.50	1
6	'2023-01-21'	Item 1	3.00	2
7	'2023-01-25'	Item 1	1.00	1
8	'2023-01-26'	Item 2	2.00	1

Order	Date	Item	Price	Quantity	Revenue
1	'2023-01-15'	Item 1	4.50	1	4.50 * 1 = 4.50
2	'2023-01-16'	Item 2	3.00	2	3.00 * 2 = 6.00
3	'2023-01-17'	Item 3	1.00	3	3.00 * 2 = 6.00
4	'2023-01-18'	Item 4	4.80	1	4.50 * 1 = 4.80
5	'2023-01-19'	Item 5	1.50	1	3.00 * 2 = 6.00
6	'2023-01-20'	Item 1	3.00	2	3.00 * 2 = 6.00
7	'2023-01-25'	Item 1	1.50	1	3.00 * 6 = 5.50
8	'2023-01-25'	Item 2	2.00	1	3.50 * 1 = 2.50

```
# Pandas performs element-wise operations automatically. No loops needed.
df['Revenue'] = df['Price'] * df['Quantity']
```

Phase 5: Asking Targeted Business Questions

A large dataset is only useful if you can efficiently find the needle in the haystack. We need methods to select data based on labels, position, and conditions.

Key Concept

First, we set the 'Date' as the index (`df.set_index('Date')`) to make time-based lookups fast and intuitive.

Question: What were the sales on October 1st?

Method: Label-based selection with `.loc`

```
df.loc['2023-10-01']
```

Date	OrderID	Item	Price	Quantity	Revenue
2023-10-01	100	Latte	12.50	1	12.50
2023-10-01	200	Espresso	19.00	3	15.00
2023-10-01	300	Muffin	9.00	1	10.50

Matches the index label exactly

Question: Show me the first 5 transactions.

Method: Position-based selection with `.iloc`

```
df.iloc[0:5]
```

Selects the first 5 rows by position (0 to 4)

Date	OrderID	Item	Price	Quantity	Revenue
2023-09-28	100	Coffee	8.50	1	8.50
2023-09-29	200	Latte	12.50	2	15.00
2023-10-30	400	Muffin	M.00	2	16.00
2023-10-01	500	Espresso	13.00	3	15.00
2023-10-01	300	Tea	9.00	1	13.00

Question: Which were our high-value orders over $10?

Method: Conditional selection with `.query`

```
df.query("Revenue > 10")
```

Date	OrderID	Item	Price	Quantity	Revenue
2023-09-28	100	Latte	12.50	2	12.50
2023-10-29	200	Espresso	25.00	3	15.00
2023-10-30	400	Muffin	11.25	4	11.25
2023-10-01	300	Coffee	10.50	1	10.50

Filters for rows where 'Revenue' > 10

Phase 5 (Cont.): Applying Custom Business Logic to Categorize Data

Simple queries are not enough. We often need to segment our data based on custom rules, such as identifying ‘High Value’ customers or orders, to inform marketing or operational strategies.

The ‘How’

We want to create a new ‘Category’ column, labeling any order with revenue over $10 as ‘High’, and all others as ‘Standard’.

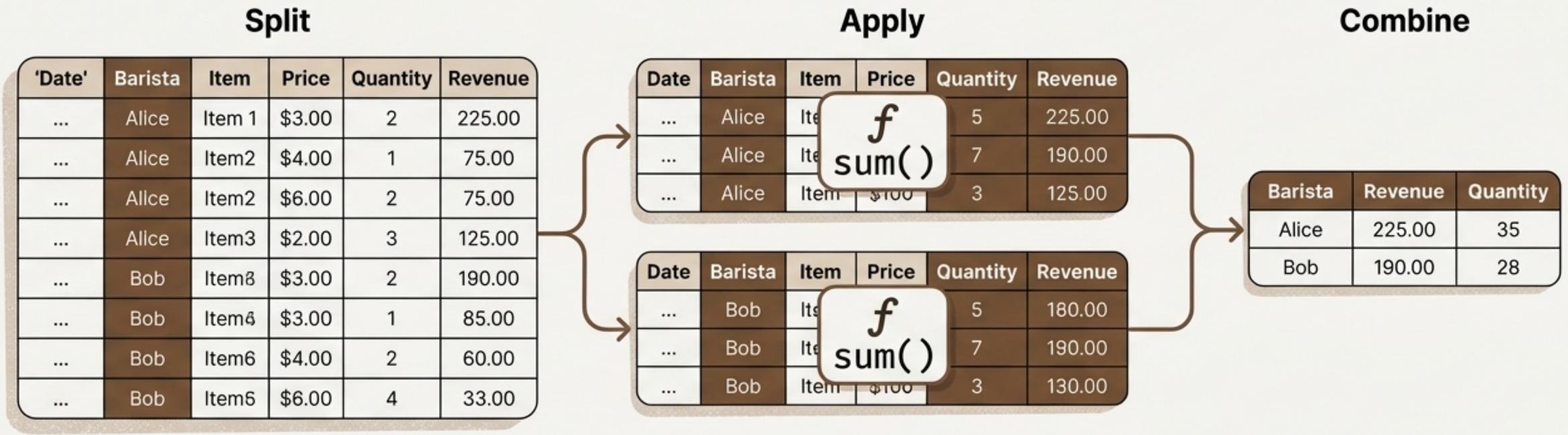
Order	Date	Item	Price	Quantity	Revenue	Category
1	2024-01-21	Product 1	12.50	2	12.50	High
2	2024-02-22	Product 2	15.00	2	15.00	High
3	2024-03-22	Product 2	11.50	3	11.25	High
4	2024-04-22	Product 3	10.00	1	9.00	Std
5	2024-04-22	Product 4	8.50	1	8.50	Std
6	2024-06-31	Product 5	10.00	2	10.00	Std

```
# A lambda is a small, anonymous function we define on the fly.
# .apply() runs this function on every row of the 'Revenue' column.
df['Category'] = df['Revenue'].apply(
    lambda x: 'High' if x > 10 else 'Std'
)
```

Phase 6: Answering “Who is Our Top-Performing Barista?” with GroupBy

To assess staff performance, we need to move from individual transaction data to aggregated summaries. The `groupby` operation is the cornerstone of this type of analysis.

The Split-Apply-Combine Strategy




```
# The core of Pandas analysis: group data by a category, # select the columns to analyze, and apply an aggregation.
performance_report = df.groupby('Barista')[['Revenue', 'Quantity']].sum()
```

Phase 6 (Cont.): Uncovering Deeper Patterns with Pivot Tables

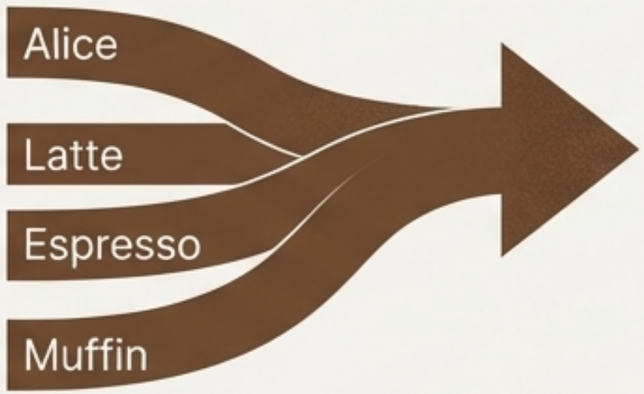
Why: An aggregate sum is good, but a breakdown of performance by product category is better. A pivot table reshapes the data from a long list into a wide grid, making complex comparisons intuitive.

Reshaping the Data for Side-by-Side Comparison

- **Goal:** We want to see how many of each item each barista sold.

FROM (Long Format)

Barista	Item	Quantity
Alice	Latte	2
Bob	Espresso	3
Alice	Muffin	1
Bob	Latte	1
Alice	Latte	1
Bob	Muffin	0
Alice	Espresso	0



TO (Wide Format)

Instantly clear: Bob sells all the Espressos while Alice moves the Muffins.

	Alice	Bob
Latte	3	1
Espresso	0	3
Muffin	1	0

```
# Create a grid to compare sales patterns
df.pivot_table(index='Item', columns='Barista', values='Quantity', aggfunc='sum')
```

Phase 7: Analyzing Business Performance Over Time

A single day's performance can be misleading. To understand the true health of the business, we must analyze trends, smoothing out daily volatility to see the real growth trajectory.

Key Techniques for Trend Analysis

Resampling

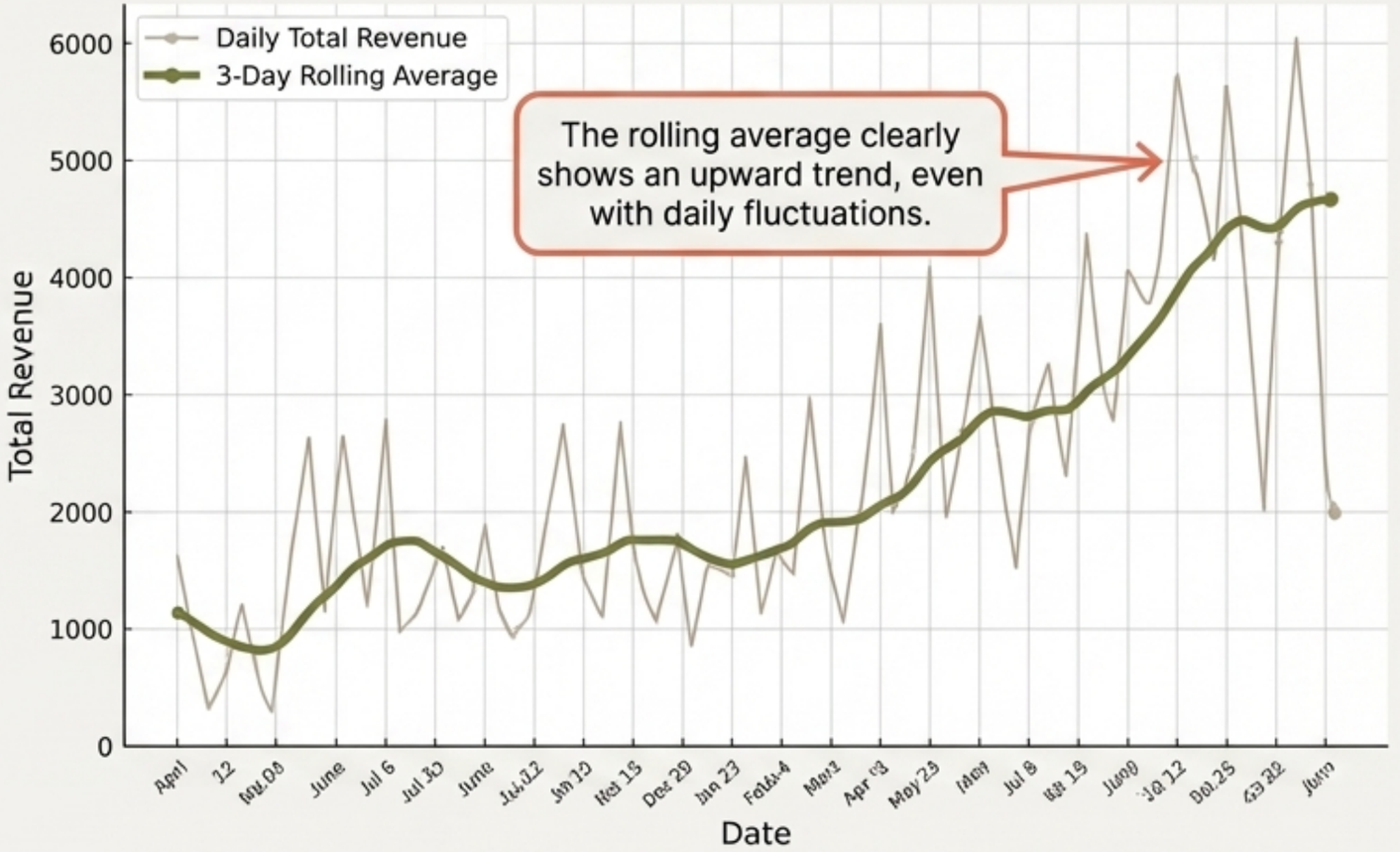
First, we aggregate our data from individual transactions into daily totals to see the big picture.

```
df['Revenue'].resample('D').sum()
```

Rolling Average

We then calculate a moving average to smooth out the noise and reveal the underlying trend.

```
.rolling(window=3).mean()
```



The Transformation: PyCafe's New Business Intelligence Report

By applying a systematic Pandas workflow, we transformed a messy collection of receipts into a clear, data-driven view of the business.

Profitability Analysis

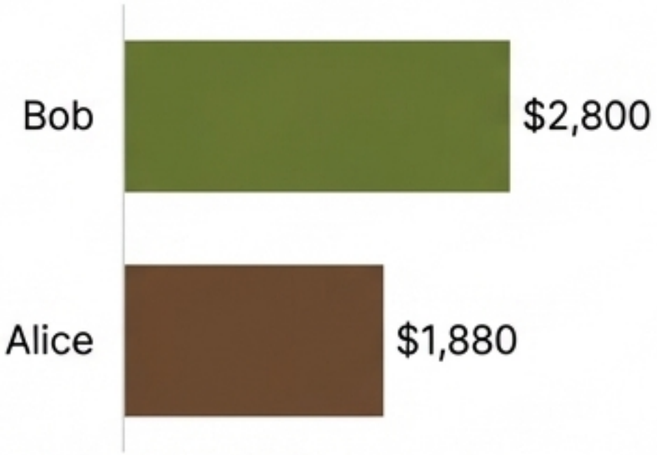
Are we making money?

Total Revenue

$4,680.50

Staff Performance

Who is the top-performing barista?



Product Insights

What are our sales patterns?

	Alice	Bob
Latte	150	0
Espresso	270	300
Muffin	100	0

The PyCafe Playbook: Building a Reusable Data Cleaning Pipeline

Why: A great analysis is repeatable. Instead of re-writing cleaning steps every time, we can chain them into a single, elegant pipeline using Pandas' `.pipe()` method. This makes our code cleaner, more readable, and less prone to errors.

Cleaning Function

```python
def clean_data(df):
    # A function that contains our cleaning steps
    df = df.drop_duplicates(subset=['OrderID'])
    df['Quantity'] = df['Quantity'].rename('Quantity')
    # ... other steps ...
    return df
```

Feature Engineering Function

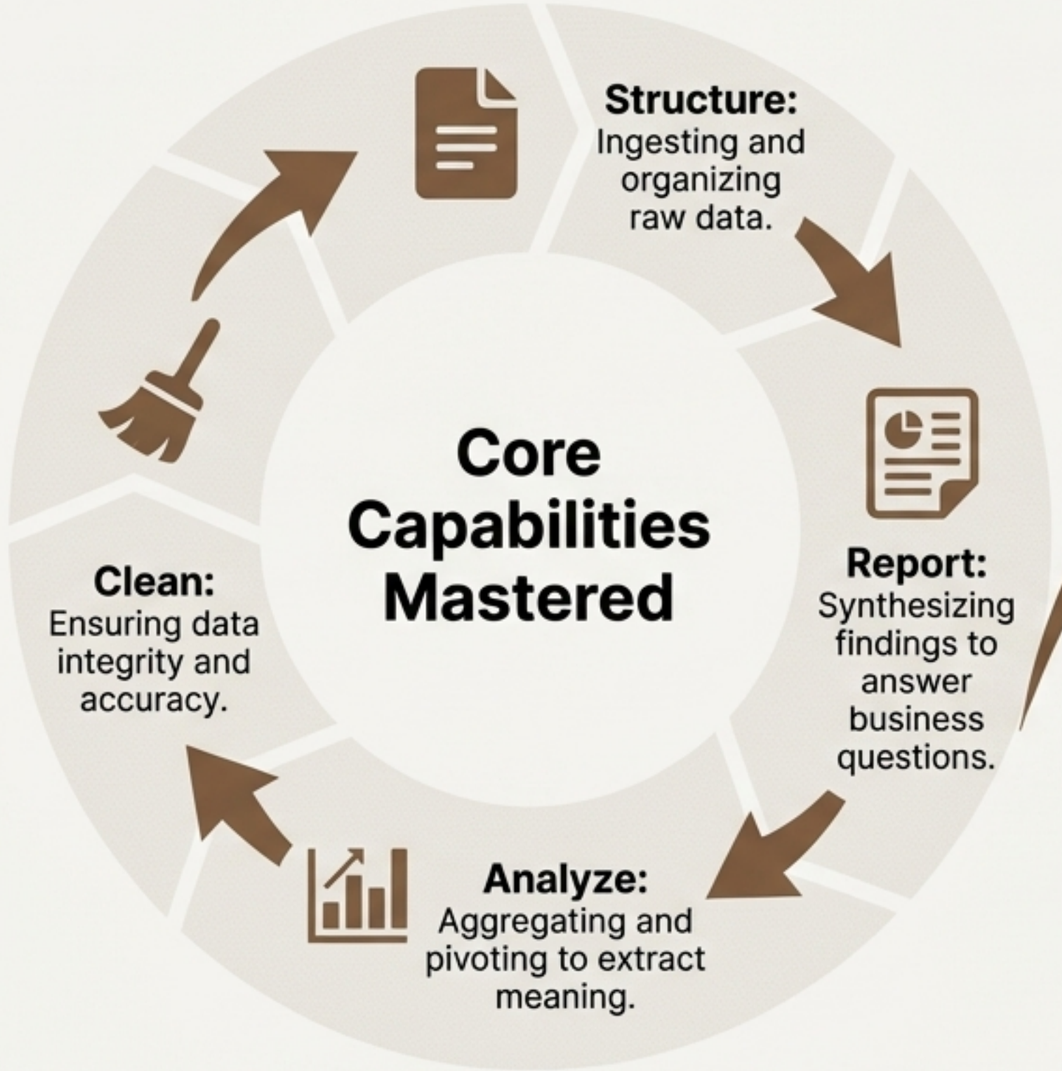
```python
def calculate_revenue(df):
    # A function for feature engineering
    df['Revenue'] = df['Price'] * df['Quantity']
    return df
```

The Final Elegant Pipeline

```python
# The elegant pipeline
final_df = (pd.read_csv('new_receipts.csv')
            .pipe(clean_data)
            .pipe(calculate_revenue)
           )
```

This approach turns a multi-step process into a documented, reusable asset for PyCafe.

Your Data Journey Continues



Next Steps: From Numbers to Narratives

The analysis we've completed provides the essential numbers. The next step is to communicate these findings visually.

**Coming Soon*: 'Visualizing the PyCafe Insights with Seaborn," where we will turn our data tables and reports into compelling charts and graphs.