Pandas for Data Science

readbytes

Pandas for Data Science

Guide to Data Manipulation and Analysis with Python

readbytes.github.io

2025-07-28

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Pandas

1 Introduction to Pandas

1.1 What is Pandas? Overview and History

Pandas is one of the most popular and powerful open-source libraries in the Python ecosystem designed for data manipulation and analysis. It provides easy-to-use, flexible, and high-performance data structures—primarily **Series** and **DataFrame**—that enable data scientists, analysts, and developers to efficiently handle, clean, transform, and analyze structured data. Whether you are working with time series, tabular data, or heterogeneous datasets, Pandas offers a comprehensive toolkit that simplifies these complex tasks.

1.1.1 Purpose and Significance of Pandas in Data Science

At its core, Pandas was created to fill a gap in Python's data processing capabilities. Before Pandas, Python lacked a robust and user-friendly library to work seamlessly with labeled data and to perform typical data science workflows such as filtering, aggregation, reshaping, and joining datasets. Pandas addresses these challenges by providing intuitive data structures that mirror the way analysts think about data—tables with rows and columns—while supporting a wide variety of data types.

The significance of Pandas in data science cannot be overstated. It serves as the backbone for data preprocessing, exploratory data analysis (EDA), and feature engineering, which are foundational steps in any data science or machine learning pipeline. Pandas integrates well with other scientific computing libraries such as NumPy, SciPy, Matplotlib, and scikit-learn, making it a vital part of the broader Python data ecosystem. Its versatility allows users to work efficiently with datasets of all sizes, from small CSV files to large time-series data.

1.1.2 Origins and Evolution

Pandas was created in 2008 by Wes McKinney while he was working at AQR Capital Management, a quantitative investment firm. McKinney needed a flexible and high-performance tool to perform quantitative analysis on financial data but found the existing Python tools inadequate for his needs. Inspired by the data manipulation capabilities of R's data frames and Excel, he developed Pandas as an open-source library to bring similar functionality to Python users.

Since its inception, Pandas has rapidly evolved and grown in popularity. It has become the standard library for data analysis in Python, supported by a vibrant community that continuously improves its features and performance. New releases often include optimizations, new functionalities, and better integration with emerging data tools.

1.1.3 How Pandas Fits into the Python Data Ecosystem

Pandas is often described as the "Swiss Army knife" for data scientists working in Python. It builds upon NumPy's numerical arrays by adding row and column labels, which makes data more accessible and meaningful. This labeled data structure is essential for real-world data analysis, where datasets often come with heterogeneous and missing data.

Furthermore, Pandas provides a seamless interface for reading and writing data to multiple file formats such as CSV, Excel, SQL databases, and JSON, allowing users to move data easily between different environments. Combined with Python's growing ecosystem of libraries for machine learning, visualization, and statistical analysis, Pandas plays a central role in enabling end-to-end data workflows.

1.2 Installing Pandas and Setting Up the Environment

Before diving into data manipulation with Pandas, you need to install the library and set up a suitable development environment. This section walks you through the step-by-step process of installing Pandas, choosing a coding environment, and verifying that everything is working correctly.

1.2.1 Step 1: Installing Pandas

Pandas can be installed easily using either **pip** or **conda**, two of the most popular Python package managers. You should already have Python installed on your system. If not, visit python.org to download and install it.

Using pip

Open your terminal or command prompt and run the following command:

```
pip install pandas
```

This command will download and install the latest stable version of Pandas along with its dependencies like NumPy.

Using conda

If you use the Anaconda or Miniconda distribution (highly recommended for data science), install Pandas with:

```
conda install pandas
```

Conda manages packages and dependencies efficiently and is often preferred for data science projects.

1.2.2 Step 2: Setting Up Your Development Environment

There are several options to write and run Python code with Pandas:

Jupyter Notebook

Jupyter Notebook is an interactive web-based environment popular among data scientists.

- To install Jupyter, run:
  ```
  pip install notebook
  ```
  or if using conda:
  ```
  conda install notebook
  ```
- Launch a notebook by typing:
  ```
  jupyter notebook
  ```
  This will open a new tab in your browser where you can create `.ipynb` notebooks and experiment with Pandas interactively.

Integrated Development Environments (IDEs)

You can also use IDEs like **VS Code**, **PyCharm**, or **Spyder**.

- **VS Code** is free and lightweight. Install the Python extension for VS Code for code editing, debugging, and integrated terminal access.
- **PyCharm** offers a powerful data science edition with advanced features.
- **Spyder** is bundled with Anaconda and tailored for scientific computing.

1.2.3 Step 3: Verifying the Installation

After installation, verify Pandas is ready to use. Open a Python shell or your chosen environment and run:

```
import pandas as pd
print(pd.__version__)
```

If no errors appear and the version number prints, Pandas is installed successfully.

1.2.4 Troubleshooting Common Issues

- **pip command not found:** Make sure Python and pip are added to your system's PATH variable. Try running `python -m pip install pandas` instead.
- **Version conflicts:** If you have multiple Python versions, confirm you are installing Pandas for the right Python interpreter.

- **Missing dependencies:** Sometimes NumPy or other packages might fail to install. Use `conda` if possible, as it handles dependencies more smoothly.
- **Jupyter not launching:** Try reinstalling Jupyter or updating your browser.

Setting up Pandas and a friendly environment is the first step toward effective data analysis with Python. Once installed, you're ready to start creating and manipulating data with Pandas' powerful tools!

1.3 Your First Pandas Program: Creating a Series and DataFrame

Now that you have Pandas installed and your environment set up, it's time to write your first Pandas program! This section will guide you through creating two fundamental Pandas data structures: a **Series** and a **DataFrame**. By working with these simple examples, you'll get a hands-on feel for how Pandas operates.

1.3.1 Importing Pandas

Before you can use Pandas, you need to import it into your Python script or notebook. By convention, Pandas is imported with the alias `pd` for convenience:

```
import pandas as pd
```

1.3.2 Creating a Pandas Series

A **Series** is essentially a one-dimensional labeled array that can hold data of any type (numbers, strings, etc.). Think of it as a single column of data with an index.

Here's how to create a Series:

Full runnable code:

```
import pandas as pd

# Create a Series from a list of numbers
numbers = pd.Series([10, 20, 30, 40, 50])

print(numbers)
```

Output:

```
0    10
```

```
1    20
2    30
3    40
4    50
dtype: int64
```

Notice how Pandas automatically assigns an integer index starting at 0. You can also specify custom labels for the index if you need.

Try changing the list values or adding your own labels to experiment with how the Series behaves!

1.3.3 Creating a Pandas DataFrame

A **DataFrame** is a two-dimensional labeled data structure—think of it as a spreadsheet or SQL table. It consists of rows and columns, where each column can be a different data type.

Here's an example of creating a DataFrame from a dictionary:

Full runnable code:

```python
import pandas as pd

# Create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

print(df)
```

Output:

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

Each key in the dictionary becomes a column, and the lists provide the column data. Pandas automatically assigns a default integer index here as well.

1.3.4 Encourage Experimentation

Feel free to modify the data, add more rows or columns, or create Series and DataFrames from other data types like NumPy arrays or CSV files. The best way to learn Pandas is by playing with data and observing how these structures behave.

Try running the examples above, then:

- Change the values or labels
- Create a Series with strings or mixed types
- Add a new column to the DataFrame like `df['Salary'] = [70000, 80000, 90000]`

With these simple building blocks, you're already on your way to mastering data manipulation in Pandas!

1.4 Understanding Data Structures: Series vs DataFrame

Pandas provides two primary data structures that form the foundation for almost all data manipulation and analysis: the **Series** and the **DataFrame**. Understanding the differences between these two is essential to harness the full power of Pandas in your data science projects.

1.4.1 What is a Series?

A **Series** is a one-dimensional labeled array capable of holding any data type, such as integers, floats, strings, or Python objects. It consists of two main components:

- **Values:** The data stored in the Series.
- **Index:** A set of labels that uniquely identify each data point.

A Series can be thought of as a single column of data with an index.

When to use a Series?

- When you need to work with a single column or a sequence of values.
- When you want to perform vectorized operations on one-dimensional data.
- When representing time series or any ordered data.

Example of a Series

Full runnable code:

```python
import pandas as pd

# Series representing daily temperatures
```

```
temperatures = pd.Series([22, 25, 21, 23, 24], index=['Mon', 'Tue', 'Wed', 'Thu', 'Fri'])
print(temperatures)
```

Output:

```
Mon    22
Tue    25
Wed    21
Thu    23
Fri    24
dtype: int64
```

Here, the days of the week serve as the index, and the values represent temperatures.

1.4.2 What is a DataFrame?

A **DataFrame** is a two-dimensional labeled data structure with columns of potentially different types. You can think of it as a table or spreadsheet with rows and columns:

- **Rows:** Each row has a unique index label.
- **Columns:** Each column has a label and can hold data of a specific type.

When to use a DataFrame?

- When working with multiple variables or features organized in columns.
- When you want to store heterogeneous data types in one structure.
- When you need to perform operations across rows, columns, or subsets of the data.

Example of a DataFrame

Full runnable code:

```
import pandas as pd
# DataFrame representing students' details
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22],
    'Grade': ['A', 'B', 'A']
}

students = pd.DataFrame(data)
print(students)
```

Output:

```
      Name  Age Grade
0    Alice   25     A
1      Bob   30     B
2  Charlie   22     A
```

This DataFrame contains three columns with different data types: strings, integers, and categorical grades.

1.4.3 Key Differences Between Series and DataFrame

Feature	Series	DataFrame
Dimensions	One-dimensional	Two-dimensional
Structure	Single column with an index	Multiple columns with row & column labels
Data Types	Holds a single data type (or mixed but usually homogeneous)	Can hold multiple data types across columns
Use Cases	Single variable analysis, time series	Tabular data, multi-variable datasets
Access	Access values by index label	Access rows and columns via labels

1.4.4 Basic Operations

- **Series:** You can perform element-wise operations like filtering, arithmetic, and aggregation.

```
# Filtering temperatures above 22 degrees
warm_days = temperatures[temperatures > 22]
print(warm_days)
```

- **DataFrame:** You can select columns, filter rows, group data, and perform complex transformations.

```
# Select students older than 23
older_students = students[students['Age'] > 23]
print(older_students)
```

1.4.5 Why These Data Structures Matter

Series and DataFrames are the building blocks for data workflows in Pandas. They allow you to organize, manipulate, and analyze data efficiently while retaining meaningful labels. Whether you are cleaning data, exploring patterns, or preparing features for machine learning, understanding these two structures lets you apply the right tools for the job.

By mastering Series and DataFrames, you unlock the ability to handle real-world data with flexibility and power — the essence of what makes Pandas an indispensable library for data science.

Chapter 2.

Pandas Data Structures in Depth

1. Series: Creation, Indexing, and Basic Operations
2. DataFrame: Creation from Lists, Dicts, and Arrays
3. Index Objects: Importance and Custom Indexing
4. Practical Examples: Simple Data Exploration

2 Pandas Data Structures in Depth

2.1 Series: Creation, Indexing, and Basic Operations

The **Series** is the fundamental one-dimensional labeled data structure in Pandas. It holds data of any type (integers, floats, strings, Python objects) along with an associated index that labels each element. In this section, we will explore multiple ways to create a Series, how to access and manipulate its elements through indexing and slicing, and perform basic operations like arithmetic and aggregation.

2.1.1 Creating a Series

From a List

The simplest way to create a Series is by passing a Python list:

Full runnable code:

```
import pandas as pd

data_list = [10, 20, 30, 40, 50]
series_from_list = pd.Series(data_list)
print(series_from_list)
```

Output:

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Here, Pandas automatically assigns integer indices starting from 0.

From a NumPy Array

You can also create a Series from a NumPy array:

Full runnable code:

```
import numpy as np

data_array = np.array([1.5, 2.5, 3.5, 4.5])
series_from_array = pd.Series(data_array)
print(series_from_array)
```

Output:

```
0    1.5
1    2.5
2    3.5
3    4.5
dtype: float64
```

From a Dictionary

When creating a Series from a dictionary, the keys become the index labels and the values become the Series data:

```
import pandas as pd

data_dict = {'a': 100, 'b': 200, 'c': 300}
series_from_dict = pd.Series(data_dict)
print(series_from_dict)
```

Output:

```
a    100
b    200
c    300
dtype: int64
```

Custom Indexing on Creation

You can specify a custom index explicitly, which allows for more meaningful labels:

Full runnable code:

```
import pandas as pd
series_custom_index = pd.Series([5, 10, 15], index=['x', 'y', 'z'])
print(series_custom_index)
```

2.1.2 Indexing and Accessing Elements

You can access Series elements similarly to Python lists, but Pandas also supports label-based access using `.loc` and position-based access using `.iloc`.

Full runnable code:

```
import pandas as pd

s = pd.Series([100, 200, 300], index=['a', 'b', 'c'])

# Access by position (integer location)
print(s[0])       # Output: 100

# Access by label
print(s['b'])     # Output: 200
```

```python
# Multiple elements by labels
print(s[['a', 'c']])
```

Output:

```
a    100
c    300
dtype: int64
```

Slicing

You can slice a Series using both labels and positions:

Full runnable code:

```python
import pandas as pd

s = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
print(s[1:3])        # Slicing by position (like Python slicing)

print(s['a':'c'])    # Slicing by label (inclusive of end label)
```

2.1.3 Basic Operations on Series

Arithmetic Operations

Series support element-wise arithmetic operations. Operations align automatically based on index labels:

Full runnable code:

```python
import pandas as pd

s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['a', 'b', 'c'])

print(s1 + s2)
```

Output:

```
a    5
b    7
c    9
dtype: int64
```

If indices don't match, Pandas fills missing labels with `NaN`:

```python
s3 = pd.Series([7, 8], index=['a', 'd'])
print(s1 + s3)
```

Output:

```
a     8.0
b     NaN
c     NaN
d     NaN
dtype: float64
```

Aggregation Functions

You can apply common aggregation functions such as:

- `.sum()`: sum of all elements
- `.mean()`: average value
- `.min()` / `.max()`: minimum and maximum
- `.std()`: standard deviation

Example:

Full runnable code:

```
import pandas as pd


data = pd.Series([10, 20, 30, 40, 50])
print("Sum:", data.sum())
print("Mean:", data.mean())
print("Max:", data.max())
```

Output:

```
Sum: 150
Mean: 30.0
Max: 50
```

2.1.4 Series with Diverse Data Types

Series can hold data of different types, including strings:

Full runnable code:

```
import pandas as pd

mixed_series = pd.Series(['apple', 'banana', 'cherry'])
print(mixed_series)
```

Output:

```
0     apple
1    banana
```

```
2    cherry
dtype: object
```

2.1.5 Summary

- You can create Series from lists, arrays, and dictionaries.
- Series have labeled indices which make data access intuitive.
- Use `.loc` for label-based and `.iloc` for position-based indexing.
- Series support element-wise arithmetic operations aligned by index.
- Aggregation functions simplify statistical summaries.
- Series handle diverse data types, making them flexible for many applications.

Try creating Series with your own data and experiment with indexing and operations to build a solid foundation for your Pandas journey!

2.2 DataFrame: Creation from Lists, Dicts, and Arrays

The **DataFrame** is the most commonly used data structure in Pandas—a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled rows and columns. This section explores how to create DataFrames from different data sources including lists, dictionaries, and NumPy arrays, while highlighting how column and row labels are assigned and how the input format influences the resulting DataFrame structure.

2.2.1 Creating a DataFrame from a List of Lists

One straightforward way to create a DataFrame is by passing a list of lists, where each inner list represents a row.

Full runnable code:

```
import pandas as pd

data = [
    [101, 'Alice', 25],
    [102, 'Bob', 30],
    [103, 'Charlie', 22]
]

df = pd.DataFrame(data, columns=['ID', 'Name', 'Age'])
print(df)
```

Output:

```
    ID     Name  Age
0  101    Alice   25
1  102      Bob   30
2  103  Charlie   22
```

- **Columns:** We explicitly assign column labels with the `columns` parameter.
- **Rows:** Since no index is provided, Pandas automatically assigns a default integer index starting at 0.

2.2.2 Creating a DataFrame from a Dictionary of Lists

More commonly, data comes in the form of a dictionary where each key represents a column, and its value is a list of column entries.

Full runnable code:

```
import pandas as pd

data_dict = {
    'ID': [201, 202, 203],
    'Name': ['David', 'Eva', 'Frank'],
    'Age': [28, 24, 27]
}

df_dict = pd.DataFrame(data_dict)
print(df_dict)
```

Output:

```
    ID   Name  Age
0  201  David   28
1  202    Eva   24
2  203  Frank   27
```

- **Column labels:** Taken directly from the dictionary keys.
- **Row labels:** Default integer index assigned if not specified.

2.2.3 Creating a DataFrame from NumPy Arrays

You can also create DataFrames from NumPy arrays, which are useful if you're working with numerical data generated or processed by NumPy.

Full runnable code:

```
import numpy as np
import pandas as pd

data_array = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

df_array = pd.DataFrame(data_array, columns=['A', 'B', 'C'])
print(df_array)
```

Output:

```
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```

- **Columns:** You need to provide column labels explicitly as NumPy arrays don't have named columns.
- **Rows:** Integer indices are automatically assigned.

2.2.4 Customizing Row Index Labels

You can provide custom row indices with the `index` parameter to assign meaningful row labels:

```
df_custom_index = pd.DataFrame(data_dict, index=['a', 'b', 'c'])
print(df_custom_index)
```

Output:

```
    ID   Name  Age
a  201  David   28
b  202    Eva   24
c  203  Frank   27
```

This is especially useful when your data rows represent identifiable entities like dates, IDs, or categories.

2.2.5 How Input Format Affects DataFrame Structure

- **List of lists:** Treated as rows; requires specifying columns for clarity.
- **Dictionary of lists:** Keys become columns; lengths of lists must be equal, or Pandas will raise an error.

- **NumPy arrays:** Numeric data without labels, so column and row labels need to be provided.

2.2.6 Summary

Creating DataFrames is flexible and adapts to your data's original structure:

Input Type	Structure Created	Labeling
List of lists	Rows from each inner list	Columns need to be specified manually
Dictionary of lists	Columns from dict keys	Rows get default integer index unless specified
NumPy arrays	Numeric matrix	Requires explicit column and optional row labels

Experiment with different inputs and labels to see how the DataFrame shape and labeling behave. Mastery of these creation methods lays the foundation for effective data handling in Pandas.

2.3 Index Objects: Importance and Custom Indexing

In pandas, **Index objects** are the backbone of how data is aligned and retrieved. Whether working with a `Series` or a `DataFrame`, the index provides a **label-based mechanism** to access rows and ensures consistent data alignment across operations such as joins, filters, or merges.

An `Index` behaves like an immutable array or set of labels. It can hold any hashable Python object (strings, numbers, timestamps, etc.), and it's optimized for **fast lookups** and **automatic alignment** of data. For example, when performing arithmetic between two Series or DataFrames, pandas aligns values using their indexes rather than their physical position.

Let's explore the creation and customization of Index objects.

Creating a Custom Index

Full runnable code:

```
import pandas as pd

data = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
print(data)
```

Output:

```
a    100
b    200
c    300
dtype: int64
```

Here, we assigned a custom index (`'a'`, `'b'`, `'c'`) instead of the default integer index. This allows us to access values using labels:

```
print(data['b'])  # Output: 200
```

Setting and Resetting the Index in DataFrames

In `DataFrames`, indexes can be customized using the `.set_index()` method, and reverted with `.reset_index()`.

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})
df = df.set_index('Name')
print(df)
```

Output:

```
         Age
Name
Alice     25
Bob       30
Charlie   35
```

Now, `'Name'` is the index. To bring it back as a regular column:

```
df = df.reset_index()
print(df)
```

Index Uniqueness

Indexes **do not have to be unique**, but unique indexes enable faster and more predictable lookups. When duplicate labels exist, selecting a label returns **multiple rows**.

Full runnable code:

```
import pandas as pd
s = pd.Series([1, 2, 3], index=['x', 'x', 'y'])
print(s['x'])  # Returns a Series with both 'x' entries
```

You can check for uniqueness:

```
print(s.index.is_unique)  # Output: False
```

Introduction to MultiIndex (Hierarchical Indexing)

A **MultiIndex** allows more than one index level, enabling complex data structures like pivot tables.

Full runnable code:

```
import pandas as pd
arrays = [
    ['Math', 'Math', 'Science', 'Science'],
    ['Alice', 'Bob', 'Alice', 'Bob']
]
index = pd.MultiIndex.from_arrays(arrays, names=['Subject', 'Student'])
scores = pd.Series([90, 85, 95, 80], index=index)
print(scores)
```

Output:

```
Subject  Student
Math     Alice      90
         Bob        85
Science  Alice      95
         Bob        80
dtype: int64
```

This format makes it easy to slice data across multiple dimensions:

```
print(scores['Math'])  # All Math scores
print(scores[:, 'Alice'])  # All scores for Alice
```

2.3.1 Summary

Pandas Index objects provide powerful capabilities for data labeling, alignment, and retrieval. With features like custom indexing, label-based access, index resetting, and multi-level indexes, you gain precise control over your data structure—an essential skill for effective data analysis.

2.4 Practical Examples: Simple Data Exploration

In this section, we'll put together the foundational knowledge you've gained about `Series`, `DataFrame`, and indexing by walking through simple, hands-on examples of data exploration. These small but powerful techniques are commonly used at the start of any data analysis process.

We'll begin by creating a small dataset and then demonstrate how to explore and manipulate it using pandas.

Step 1: Create a Sample Dataset

Let's create a `DataFrame` containing fictional student exam data:

Full runnable code:

```python
import pandas as pd

data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Subject': ['Math', 'Science', 'Math', 'History', 'Science'],
    'Score': [88, 92, 95, 70, 85],
    'Passed': [True, True, True, False, True]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
  Student  Subject  Score  Passed
0   Alice     Math     88    True
1     Bob  Science     92    True
2 Charlie     Math     95    True
3   David  History     70   False
4     Eva  Science     85    True
```

Step 2: Basic Data Exploration

Use pandas methods to quickly understand the structure and contents of your data.

Full runnable code:

```python
import pandas as pd

data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Subject': ['Math', 'Science', 'Math', 'History', 'Science'],
    'Score': [88, 92, 95, 70, 85],
    'Passed': [True, True, True, False, True]
}

df = pd.DataFrame(data)

# First few rows
print(df.head())
```

```python
# Summary of column types and non-null values
print(df.info())
```

```python
# Statistical summary of numerical columns
print(df.describe())
```

```
# Unique values in a column
print(df['Subject'].unique())
```

Step 3: Accessing Data with Series

You can extract a column as a `Series` for further inspection:

```
scores = df['Score']
print(type(scores))  # <class 'pandas.core.series.Series'>
print(scores.mean())  # Average score
```

You can also filter rows using boolean conditions:

```
print(df[df['Score'] > 90])  # Students scoring above 90
```

Step 4: Custom Indexing

To make lookups and sorting easier, set a more meaningful index:

```
df_indexed = df.set_index('Student')
print(df_indexed)
```

Output:

```
         Subject  Score  Passed
Student
Alice       Math     88    True
Bob      Science     92    True
Charlie     Math     95    True
David    History     70   False
Eva      Science     85    True
```

Now you can access rows by student name:

```
print(df_indexed.loc['Charlie'])
```

You can always reset the index if needed:

```
df_reset = df_indexed.reset_index()
print(df_reset.head())
```

2.4.1 Summary

These basic operations—viewing, summarizing, filtering, and indexing—form the foundation of data exploration with pandas. With just a few lines of code, you can gain immediate insights into your dataset and prepare it for deeper analysis. The more you practice these techniques, the more natural and efficient your data analysis will become.

Chapter 3.

Data Selection and Indexing

3 Data Selection and Indexing

3.1 Selecting Columns and Rows with `.loc` and `.iloc`

In pandas, selecting data efficiently is essential for analysis, filtering, and transformation. Two of the most powerful and commonly used tools for this task are `.loc` and `.iloc`.

These accessors allow you to retrieve rows and columns from a DataFrame or Series using either **labels** or **integer positions**. Let's dive into what each does and how to use them effectively.

3.1.1 `.loc` Label-Based Selection

The `.loc` accessor selects data by **label** — that is, the name of the row and/or column.

Full runnable code:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Toronto', 'London']
}
df = pd.DataFrame(data, index=['a', 'b', 'c'])
```

Select a Single Row by Label

```
print(df.loc['a'])
```

Output:

```
Name     Alice
Age         25
City  New York
Name: a, dtype: object
```

Select Multiple Rows by Label

```
print(df.loc[['a', 'c']])
```

Select a Single Column by Label

```
print(df.loc[:, 'Name'])
```

Select Multiple Columns

```
print(df.loc[:, ['Name', 'City']])
```

Slice Rows by Label

```
print(df.loc['a':'b'])  # Includes both 'a' and 'b'
```

3.1.2 `.iloc` Position-Based Selection

The `.iloc` accessor selects data by **integer position**, similar to how Python lists and arrays work.

Select a Single Row by Position

```
print(df.iloc[0])  # First row
```

Select Multiple Rows

```
print(df.iloc[[0, 2]])  # First and third rows
```

Select a Range of Rows

```
print(df.iloc[0:2])  # First and second rows
```

Select a Specific Cell (row 1, column 2)

```
print(df.iloc[1, 2])  # Output: Toronto
```

Select Rows and Columns by Position

```
print(df.iloc[0:2, 1:3])  # Rows 0-1, Columns 1-2
```

3.1.3 When to Use `.loc` vs `.iloc`

Use Case	Method	Example
Select by **row/column name**	`.loc`	`df.loc['a', 'Name']`
Select by **row/column position**	`.iloc`	`df.iloc[0, 0]`
Slice by **label range**	`.loc`	`df.loc['a':'b']`
Slice by **integer range**	`.iloc`	`df.iloc[0:2]`

Always use `.loc` when you're dealing with **named indexes or columns** (like "Name" or "a"), and `.iloc` when you're dealing with **numerical positions** (like 0 or 1).

3.1.4 Practical Tip

Sometimes, it's easy to confuse the two. Here's a quick comparison to clarify:

```
# First row, 'Name' column
df.loc['a', 'Name']    # Label-based
df.iloc[0, 0]          # Position-based
```

3.1.5 Summary

- Use `.loc` for label-based selection (row and column names).
- Use `.iloc` for position-based selection (row and column numbers).
- Both support slicing, lists of indexes, and multidimensional selection.
- Choosing the right accessor keeps your code clean, predictable, and efficient.

Mastering `.loc` and `.iloc` is a cornerstone of working effectively with pandas. They provide precision and flexibility in selecting data from complex tables—skills you'll use constantly in real-world data analysis.

3.2 Boolean Indexing and Conditional Filtering

One of the most powerful features in pandas is the ability to filter data using **boolean indexing**. This allows you to select rows based on **conditions**, such as finding all rows where a value exceeds a threshold or where multiple criteria are met. Boolean indexing is both intuitive and highly expressive, making it essential for everyday data manipulation.

3.2.1 Basic Boolean Filtering

To begin, let's create a simple `DataFrame`:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [25, 30, 35, 40, 28],
    'Score': [88, 92, 95, 70, 85],
    'Passed': [True, True, True, False, True]
}

df = pd.DataFrame(data)
```

Filter: Age 30

You can use a single condition to filter rows:

```
filtered = df[df['Age'] > 30]
print(filtered)
```

Output:

```
      Name  Age  Score  Passed
2  Charlie   35     95    True
3    David   40     70   False
```

The condition `df['Age'] > 30` returns a **boolean Series**:

```
print(df['Age'] > 30)
```

Output:

```
0    False
1    False
2     True
3     True
4    False
Name: Age, dtype: bool
```

This boolean Series is then used to filter the original `DataFrame`.

3.2.2 Combining Multiple Conditions

Use the **bitwise operators** `&` (and), `|` (or), and `~` (not) to combine conditions. **Enclose each condition in parentheses** to avoid precedence errors.

Filter: Age 30 AND Score 80

```
df_filtered = df[(df['Age'] > 30) & (df['Score'] > 80)]
print(df_filtered)
```

Output:

```
      Name  Age  Score  Passed
2  Charlie   35     95    True
```

Filter: Passed OR Score 90

```
df_filtered = df[(df['Passed'] == True) | (df['Score'] >= 90)]
print(df_filtered)
```

3.2.3 Filtering Using Equality and Membership

Filter: Name is "Alice"

```
print(df[df['Name'] == 'Alice'])
```

Filter: Name in list

```
print(df[df['Name'].isin(['Alice', 'Eva'])])
```

Filter: Name does not appear in list

```
print(df[~df['Name'].isin(['Alice', 'Eva'])])
```

3.2.4 Using `.query()` for Simpler Syntax

For more readable code, pandas offers the `.query()` method:

```
print(df.query("Age > 30 and Score > 80"))
```

This returns the same result as using boolean indexing with `&`.

3.2.5 Practical Tip

You can assign the filtered result to a new variable or use it to modify values:

```
# Give a bonus point to students with Score > 90
df.loc[df['Score'] > 90, 'Score'] += 5
```

3.2.6 Summary

Boolean indexing allows you to filter data based on one or more conditions using intuitive expressions. Whether you're selecting high scores, filtering by name, or combining multiple filters, mastering this technique is crucial for effective data analysis. Practice combining filters and try out `.query()` for more readable syntax in complex cases.

3.3 Setting and Resetting Indexes

In pandas, every `DataFrame` has an **index**, which serves as the row identifier. By default, pandas assigns a sequential integer index starting from 0. However, in real-world data manipulation, it's often useful to **set one or more meaningful columns as the index** — especially when the column values are unique identifiers like names, dates, or IDs.

This section explains how to set and reset indexes, and how it affects data operations.

3.3.1 Setting a Column as Index

You can set a column as the index using the `.set_index()` method. This can improve data lookup efficiency and make your DataFrame more intuitive.

Full runnable code:

```
import pandas as pd

data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'David'],
    'Subject': ['Math', 'Science', 'Math', 'History'],
    'Score': [88, 92, 95, 70]
}

df = pd.DataFrame(data)
df_indexed = df.set_index('Student')
print(df_indexed)
```

Output:

```
         Subject  Score
Student
Alice       Math     88
Bob      Science     92
Charlie     Math     95
David    History     70
```

Now, `"Student"` is the index, and you can access rows by name:

```
print(df_indexed.loc['Charlie'])  # Outputs row for Charlie
```

You can set the index **in-place** by passing `inplace=True`:

```
df.set_index('Student', inplace=True)
```

3.3.2 Resetting the Index

To return the index to the default integer format, use `.reset_index()`:

```
df_reset = df_indexed.reset_index()
print(df_reset)
```

Output:

```
  Student  Subject  Score
0   Alice     Math     88
1     Bob  Science     92
2 Charlie     Math     95
3   David  History     70
```

By default, `reset_index()` **moves the index back to a column**. If you need to discard the current index completely, set `drop=True`:

```
df_reset = df_indexed.reset_index(drop=True)
```

3.3.3 When to Set or Reset the Index

Use `.set_index()` when:

- You want to perform fast lookups using meaningful identifiers.
- You're preparing data for grouped operations or pivot tables.
- You want cleaner output when printing or exporting.

Use `.reset_index()` when:

- You need to flatten a multi-level index (e.g., after grouping).
- You want to modify or sort index values as columns.
- You're preparing data for export where default indexing is preferred.

3.3.4 Common Pitfall

If you forget to set `inplace=True` or assign the result to a new variable, the operation has no effect:

```
df.set_index('Student')  # Won't change df unless assigned
```

Always either:

```
df = df.set_index('Student')
```

or:

```
df.set_index('Student', inplace=True)
```

3.3.5 Summary

Setting and resetting indexes is a fundamental pandas technique that can streamline data access and improve clarity. Use meaningful columns as indexes when they uniquely identify rows, and reset the index when you need to simplify or restructure the DataFrame.

3.4 Practical Examples: Filtering Datasets

Now that you've learned about selecting data with `.loc` and `.iloc`, filtering with boolean conditions, and manipulating indexes, it's time to apply these concepts in practice. This section walks through small, end-to-end examples using a sample dataset. You'll learn how to combine selection techniques and filter subsets of data for real-world exploration.

Let's start with a simple dataset of students and their exam results.

3.4.1 Sample Dataset

```
import pandas as pd

data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Subject': ['Math', 'Science', 'Math', 'History', 'Science'],
    'Score': [88, 92, 95, 70, 85],
    'Passed': [True, True, True, False, True]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
  Student  Subject  Score  Passed
0   Alice     Math     88    True
1     Bob  Science     92    True
2 Charlie     Math     95    True
3   David  History     70   False
4     Eva  Science     85    True
```

3.4.2 Example 1: Filter Students Who Passed Math

```
math_passed = df[(df['Subject'] == 'Math') & (df['Passed'] == True)]
print(math_passed)
```

Output:

```
  Student Subject  Score  Passed
0   Alice    Math     88    True
2 Charlie    Math     95    True
```

This uses **boolean indexing** with multiple conditions and the `&` operator.

3.4.3 Example 2: Set 'Student' as Index and Retrieve by Name

```
df_indexed = df.set_index('Student')
print(df_indexed.loc['Charlie'])
```

Output:

```
Subject     Math
Score         95
Passed      True
Name: Charlie, dtype: object
```

Setting an index makes name-based lookups easy with `.loc`.

3.4.4 Example 3: Students with Score Between 85 and 95

```
mid_scores = df[(df['Score'] >= 85) & (df['Score'] <= 95)]
print(mid_scores[['Student', 'Score']])
```

Output:

```
  Student  Score
0   Alice     88
1     Bob     92
2 Charlie     95
4     Eva     85
```

You can use **range conditions** and subset to only relevant columns.

3.4.5 Example 4: Reset Index After Filtering

```
science_df = df[df['Subject'] == 'Science'].set_index('Student')
print(science_df)

# Reset to prepare for export or reindexing
science_reset = science_df.reset_index()
print(science_reset)
```

This workflow is common: **filter** → **set index** → **analyze** → **reset index**.

3.4.6 Practice Tip

Try combining these steps:

- Filter students who passed and scored above 90.
- Set `"Subject"` as index and retrieve all rows for `"Science"`.
- Use `.loc` to access a specific student and check their score.

3.4.7 Summary

These examples show how selection, boolean filtering, and index manipulation work together in real-world tasks. With a few lines of code, you can isolate meaningful subsets and explore your data with precision. Practice mixing `.loc`, conditions, and indexing to become fluent in pandas filtering.

Chapter 4.

Data Cleaning and Preparation

1. Handling Missing Data: `isna()`, `fillna()`, `dropna()`
2. Data Type Conversion and Optimization
3. Renaming Columns and Reordering
4. Practical Examples: Cleaning Real-World Datasets

4 Data Cleaning and Preparation

4.1 Handling Missing Data: `isna()`, `fillna()`, `dropna()`

Real-world datasets often contain **missing or incomplete data**—whether due to user omission, system error, or inconsistencies during data collection. Handling missing values is a **critical first step** in data cleaning, as unaddressed nulls can lead to errors in analysis, misleading results, or failed machine learning models.

Pandas provides a powerful and flexible set of tools to detect, fill, and remove missing data. This section walks through practical strategies using `isna()`, `fillna()`, and `dropna()`.

4.1.1 Detecting Missing Values

Pandas represents missing data using the special `NaN` (Not a Number) marker, which can occur in both numeric and non-numeric columns.

Let's create a sample dataset with missing values:

Full runnable code:

```
import pandas as pd
import numpy as np

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', None],
    'Age': [25, 30, np.nan, 40, 28],
    'Score': [88, np.nan, 95, 70, np.nan]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
      Name   Age  Score
0    Alice  25.0   88.0
1      Bob  30.0    NaN
2  Charlie   NaN   95.0
3    David  40.0   70.0
4     None  28.0    NaN
```

To identify missing values, use `.isna()` or `.notna()`:

```
print(df.isna())
print(df['Age'].isna())
```

These return a boolean `DataFrame` or `Series`, where `True` indicates a missing value.

You can also count total missing values per column:

```
print(df.isna().sum())
```

4.1.2 Strategy 1: Dropping Missing Data with `dropna()`

You may choose to drop rows or columns with missing values if they are sparse or irrelevant.

Drop Rows with Any Missing Value

```
df_drop_rows = df.dropna()
print(df_drop_rows)
```

Drop Columns with Missing Values

```
df_drop_cols = df.dropna(axis=1)
print(df_drop_cols)
```

Drop Only If All Values Are Missing

```
df.dropna(how='all')
```

This approach is useful when rows or columns are mostly complete, and the missing data is minimal or unimportant.

4.1.3 Strategy 2: Filling Missing Data with `fillna()`

Often it's better to **fill** missing values using appropriate logic instead of dropping them.

Fill with a Constant Value

```
df_fill_const = df.fillna(0)
```

Fill with Column Mean or Median

```
df['Score'] = df['Score'].fillna(df['Score'].mean())
df['Age'] = df['Age'].fillna(df['Age'].median())
```

Forward Fill (`ffill`) and Backward Fill (`bfill`)

```
df_ffill = df.fillna(method='ffill')  # Fill with previous non-null value
df_bfill = df.fillna(method='bfill')  # Fill with next non-null value
```

These methods are helpful in time-series or sequential data where continuity matters.

4.1.4 Choosing the Right Strategy

Scenario	Recommended Action
Few rows missing in non-critical columns	Use `fillna()` with default or mean
Entire row or column is missing	Use `dropna()` with `how='all'`
Time series data with gaps	Use `ffill()` or `bfill()`
Sparse column with many NaNs	Consider dropping the column

4.1.5 Summary

Handling missing data is foundational to clean and reliable analysis. Use `.isna()` to detect, and then apply either `fillna()` or `dropna()` based on the context of your data. Whether you're working with small CSVs or large-scale databases, understanding these tools ensures that your analyses are robust and accurate.

4.2 Data Type Conversion and Optimization

Efficient data analysis begins with **correct and optimized data types**. In pandas, every column in a `DataFrame` has a specific data type (`dtype`) such as `int64`, `float64`, `object`, `category`, or `datetime64`. Choosing the appropriate type not only ensures correct calculations but also greatly improves **memory usage** and **performance**, especially for large datasets.

4.2.1 Why Data Types Matter

- **Memory Efficiency**: Using `int8` instead of `int64`, or `category` instead of `object`, can drastically reduce memory.
- **Speed**: Operations on numeric types or categoricals are faster than on strings (`object`).
- **Correctness**: Proper types ensure accurate sorting, comparisons, and calculations.

4.2.2 Inspecting Data Types

To check the data types of all columns:

```python
import pandas as pd

df = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Score': [88.5, 92.0, 95.0],
    'Registered': ['2023-01-01', '2023-01-02', '2023-01-03']
})

print(df.dtypes)
```

Output:

```
ID                int64
Name             object
Score           float64
Registered       object
```

Note: `Registered` is read as `object` (string) but should be a datetime type.

4.2.3 Converting Data Types with `astype()`

Use `.astype()` to manually convert types.

Convert `object` to `datetime64`:

```python
df['Registered'] = pd.to_datetime(df['Registered'])
```

Convert float to integer:

```python
df['Score'] = df['Score'].astype('int32')
```

Convert strings to category:

```python
df['Name'] = df['Name'].astype('category')
```

Convert to numeric (with error handling):

```python
df['Score'] = pd.to_numeric(df['Score'], errors='coerce')
```

4.2.4 Memory Optimization Example

Let's demonstrate memory savings using a larger dataset:

```python
import numpy as np
```

```python
df = pd.DataFrame({
    'ID': np.arange(1, 100001),
    'Gender': np.random.choice(['Male', 'Female'], size=100000),
    'Age': np.random.randint(18, 90, size=100000),
    'Income': np.random.uniform(30000, 120000, size=100000)
})

print(df.info(memory_usage='deep'))
```

Output:

```
Memory usage: ~10 MB
```

Now convert `Gender` to a category:

```python
df['Gender'] = df['Gender'].astype('category')
print(df.info(memory_usage='deep'))
```

Output:

```
Memory usage: ~6 MB
```

A ~40% memory reduction—just by converting a string column to `category`!

4.2.5 Automatic Type Inference

When loading data with `read_csv()`, pandas tries to infer types:

```python
df = pd.read_csv('data.csv')
```

But for large datasets, explicitly setting types with the `dtype` parameter is faster and safer:

```python
df = pd.read_csv('data.csv', dtype={'ID': 'int32', 'Gender': 'category'})
```

4.2.6 Summary

Correct data types are essential for both accuracy and efficiency in pandas. Use `.astype()` for manual conversions, `pd.to_datetime()` for dates, and `category` for repeated string values. For large datasets, optimizing types can drastically reduce memory usage and speed up processing—critical for scalable data workflows.

4.3 Renaming Columns and Reordering

Clean and well-organized column names are key to making a dataset easy to understand, analyze, and work with—especially when collaborating or preparing data for export. In pandas, you can **rename** columns or index labels using the `.rename()` method, and **reorder** columns using slicing or `.reindex()`.

This section demonstrates how to rename single or multiple columns, rename row indexes, and rearrange columns for clarity and consistency.

4.3.1 Renaming Columns with `.rename()`

Use `.rename()` with the `columns` argument to rename one or more column labels.

Rename a Single Column

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30],
    'City': ['New York', 'Toronto']
})

df_renamed = df.rename(columns={'City': 'Location'})
print(df_renamed)
```

Output:

```
    Name  Age  Location
0  Alice   25  New York
1    Bob   30   Toronto
```

Rename Multiple Columns

```
df_renamed = df.rename(columns={'Name': 'FullName', 'Age': 'Years'})
print(df_renamed)
```

You can also rename **index labels** using the `index` parameter:

```
df_named_index = df.rename(index={0: 'first', 1: 'second'})
print(df_named_index)
```

4.3.2 Renaming with .columns (Full Assignment)

For full renaming of all column headers at once:

```
df.columns = ['FullName', 'Years', 'Location']
print(df)
```

WARNING Be careful: this replaces **all** column names. Ensure the new list matches the number of columns.

4.3.3 Reordering Columns

You may want to reorder columns for readability, analysis workflows, or plotting. There are two common ways:

Column Slicing

```
# Move 'City' to the first column
df_reordered = df[['City', 'Name', 'Age']]
print(df_reordered)
```

Using .reindex()

```
df_reordered = df.reindex(columns=['City', 'Name', 'Age'])
```

This is especially useful when dynamically reordering columns using a list.

4.3.4 Practical Scenarios

- **Renaming** improves clarity: replace cryptic headers like `'C1'`, `'C2'` with meaningful names like `'Price'`, `'Quantity'`.
- **Reordering** improves usability: place key columns like `'ID'`, `'Date'`, or `'Name'` at the front for quick access.
- **Index renaming** helps when exporting labeled reports or aligning with external systems.

4.3.5 Summary

Use `.rename()` to rename columns or index labels in a clean and controlled way. Use slicing or `.reindex()` to reorganize column order for better visibility and usability. These small changes greatly enhance the readability and maintainability of your datasets—crucial for real-world data cleaning and collaboration.

4.4 Practical Examples: Cleaning Real-World Datasets

Now that we've covered essential data cleaning tools—handling missing data, converting data types, renaming columns, and reordering—let's bring it all together in a practical, end-to-end example.

We'll simulate a small "real-world" dataset with common issues such as missing values, incorrect data types, and inconsistent column names. Then, we'll apply pandas cleaning techniques step-by-step to transform the dataset into a clean and usable format.

4.4.1 Step 1: Simulate a Messy Dataset

Full runnable code:

```python
import pandas as pd
import numpy as np

data = {
    'Full Name': ['Alice Smith', 'Bob Jones', 'Charlie Zhang', None, 'Eva Johnson'],
    'Age': ['25', '30', np.nan, '40', '28'],
    'Score': ['88', '92', '95', 'NaN', None],
    'City ': ['New York', 'Toronto', 'London', 'Paris', 'Berlin']
}

df = pd.DataFrame(data)
print(df)
```

Issues:

- Column names are inconsistent (`'City '` has extra space).
- `'Age'` and `'Score'` are strings and contain missing values.
- A missing name (`None`) in `'Full Name'`.

4.4.2 Step 2: Clean Column Names

Standardize column names: lowercase, remove spaces.

```python
df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')
print(df.columns)
```

4.4.3 Step 3: Convert Data Types

Convert `'age'` and `'score'` to numeric:

```
df['age'] = pd.to_numeric(df['age'], errors='coerce')
df['score'] = pd.to_numeric(df['score'], errors='coerce')
print(df.dtypes)
```

Now missing and non-numeric values are converted to `NaN`.

4.4.4 Step 4: Handle Missing Data

Check how many values are missing:

```
print(df.isna().sum())
```

Drop rows where `'full_name'` is missing:

```
df = df.dropna(subset=['full_name'])
```

Fill missing values in `'score'` with the mean:

```
df['score'] = df['score'].fillna(df['score'].mean())
```

Forward-fill `'age'` if needed (e.g., in time-ordered data):

```
df['age'] = df['age'].fillna(method='ffill')
```

4.4.5 Step 5: Reorder Columns for Clarity

```
df = df[['full_name', 'age', 'score', 'city']]
```

4.4.6 Final Cleaned Dataset

```
print(df)
```

Output:

```
     full_name   age  score      city
0  Alice Smith  25.0   88.0  New York
1    Bob Jones  30.0   92.0   Toronto
```

```
2  Charlie Zhang 30.0   95.0    London
3     Eva Johnson 28.0   91.666667   Berlin
```

4.4.7 Summary

In this example, we:

- **Renamed and cleaned** column names.
- **Converted** string columns to numeric types with error handling.
- **Handled missing values** using both `dropna()` and `fillna()`.
- **Reordered columns** for better readability.

This is a **typical cleaning workflow** that forms the backbone of data preparation. You can adapt it to larger and more complex datasets by scaling the same steps—always starting with understanding the structure and issues in your data before applying fixes.

Chapter 5.

Data Transformation and Manipulation

5 Data Transformation and Manipulation

5.1 Adding, Modifying, and Removing Columns

Manipulating columns is fundamental in pandas, allowing you to **add new features**, **update existing data**, or **clean out unnecessary fields**. This section covers how to add, modify, and remove columns, along with best practices around inplace operations and method chaining.

5.1.1 Adding New Columns

You can create new columns by simple **assignment**, either with constants, lists, or expressions involving existing columns.

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'price': [100, 200, 300],
    'quantity': [1, 3, 2]
})

# Add constant column
df['discount'] = 10
print(df)
```

Output:

```
   price  quantity  discount
0    100         1        10
1    200         3        10
2    300         2        10
```

5.1.2 Adding Columns by Computation

New columns are often calculated from existing ones. For example, compute total cost:

```
df['total_cost'] = df['price'] * df['quantity'] - df['discount']
print(df)
```

Output:

```
   price  quantity  discount  total_cost
0    100         1        10          90
```

```
1    200        3       10        590
2    300        2       10        590
```

5.1.3 Modifying Existing Columns

To modify a column, assign new values directly:

```
# Apply a 10% price increase
df['price'] = df['price'] * 1.1
print(df['price'])
```

You can also use methods like `.apply()` or vectorized operations for more complex modifications.

5.1.4 Removing Columns

Remove columns with `.drop()`. By default, `.drop()` returns a new DataFrame and does **not** modify inplace unless you specify `inplace=True`.

```
# Drop the discount column (returns new DataFrame)
df_new = df.drop('discount', axis=1)
print(df_new)
```

Or modify the original DataFrame inplace:

```
df.drop('discount', axis=1, inplace=True)
print(df)
```

You can drop multiple columns by passing a list:

```
df.drop(['discount', 'quantity'], axis=1, inplace=True)
```

5.1.5 Inplace vs Non-Inplace Operations

- **Inplace=True** modifies the DataFrame directly, saving memory but making chaining impossible.
- Without `inplace=True`, methods return a **new DataFrame**, allowing chaining and safer workflows.

Example chaining (preferred for readability):

```
df = (
    df.assign(total_cost=lambda x: x.price * x.quantity)
      .drop('discount', axis=1)
)
```

```
print(df)
```

5.1.6 Summary

- **Add columns** by assignment: constants, lists, or computations.
- **Modify columns** by reassigning or applying functions.
- **Remove columns** with `.drop()`, choosing between inplace and non-inplace based on your workflow.
- Use **method chaining** to write clean, readable transformation pipelines.

Mastering these operations helps you efficiently engineer features and prepare your data for analysis or modeling.

5.2 Applying Functions: `apply()`, `map()`, `applymap()`

Pandas provides powerful methods to apply functions for transforming data: `apply()`, `map()`, and `applymap()`. Understanding their differences and use cases helps you manipulate data efficiently and write clean, expressive code.

5.2.1 Overview of Methods

Method	Used On	Applies Function To	Common Use Cases
`apply()`	DataFrame/Series	Row-wise or column-wise (DataFrame) or element-wise (Series)	Aggregations, transformations along axis
`map()`	Series	Element-wise	Value mapping, replacements, simple element-wise transformations
`applymap()`	DataFrame	Element-wise	Element-wise transformations on entire DataFrame

5.2.2 `apply()` Apply Function Along Axis or on Series

- On a **DataFrame**, `apply()` applies a function along rows (`axis=1`) or columns (`axis=0`, default).

- On a **Series**, it applies the function element-wise.

Example: Sum by Row and Column

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

# Sum across columns (default axis=0)
print(df.apply(sum))
```

Output:

```
A     6
B    15
dtype: int64
```

Sum across rows (`axis=1`):

```
print(df.apply(lambda row: row.sum(), axis=1))
```

Output:

```
0     5
1     7
2     9
dtype: int64
```

On Series:

```
s = pd.Series([1, 2, 3])
print(s.apply(lambda x: x**2))
```

Output:

```
0    1
1    4
2    9
dtype: int64
```

5.2.3 `map()` Element-wise on Series

`map()` is designed for element-wise transformations on a **Series**. It's often used for:

- Mapping values via a dictionary or Series.

- Applying functions element-wise.
- Replacing values.

```
s = pd.Series(['cat', 'dog', 'rabbit'])

# Map using dictionary
print(s.map({'cat': 'kitten', 'dog': 'puppy'}))
```

Output:

```
0    kitten
1     puppy
2       NaN
dtype: object
```

Map with a function:

```
print(s.map(lambda x: x.upper()))
```

Output:

```
0       CAT
1       DOG
2    RABBIT
dtype: object
```

5.2.4 `applymap()` Element-wise on DataFrame

`applymap()` applies a function **to every individual element** of a DataFrame.

```
df = pd.DataFrame({
    'A': [1, 4],
    'B': [2, 5]
})

print(df.applymap(lambda x: x**2))
```

Output:

```
    A   B
0   1   4
1  16  25
```

5.2.5 Performance Considerations and Best Practices

- **Vectorized operations** (e.g., `df + 1`, `df['A'] * 2`) are faster than these function-application methods.

- Use `map()` for simple element-wise operations on Series, especially for mapping or replacement.
- Use `apply()` for row/column-wise logic that can't be vectorized.
- Use `applymap()` sparingly, only when you need to apply a function to every cell in a DataFrame.
- For complex or repeated operations, consider writing custom vectorized functions using NumPy or Cython.

5.2.6 Summary

Method	Use When
`apply()`	Aggregate or transform along rows/columns or element-wise on Series
`map()`	Simple element-wise mapping on Series
`applymap()`	Element-wise on entire DataFrame

Understanding these methods lets you flexibly transform your data in pandas with clarity and efficiency.

5.3 Sorting and Ranking Data

Sorting and ranking data are essential techniques for organizing, analyzing, and summarizing datasets. Pandas provides straightforward methods like `sort_values()`, `sort_index()`, and `rank()` to help you order and rank your data flexibly and efficiently.

5.3.1 Sorting with `sort_values()`

Use `sort_values()` to sort a DataFrame by one or multiple columns.

Sort by Single Column (Ascending)

Full runnable code:

```python
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Score': [88, 92, 95, 70]
})
```

```
# Sort by Score ascending
print(df.sort_values(by='Score'))
```

Output:

```
      Name  Score
3    David     70
0    Alice     88
1      Bob     92
2  Charlie     95
```

Sort by Single Column (Descending)

```
print(df.sort_values(by='Score', ascending=False))
```

5.3.2 Sort by Multiple Columns

Sort by `Score` descending, then `Name` ascending:

```
df2 = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'Bob'],
    'Score': [88, 92, 92, 70]
})

sorted_df = df2.sort_values(by=['Score', 'Name'], ascending=[False, True])
print(sorted_df)
```

5.3.3 Handling Missing Values

By default, NaNs are placed at the end. Use `na_position` to control:

```
df3 = pd.DataFrame({'Name': ['Eva', 'Frank', 'Grace'], 'Score': [85, None, 90]})

print(df3.sort_values(by='Score', na_position='first'))
```

5.3.4 Sorting by Index with `sort_index()`

Sort rows by the DataFrame index:

```
df_indexed = df.set_index('Name')
print(df_indexed.sort_index())
```

Sort columns by their labels:

```
print(df.sort_index(axis=1))
```

5.3.5 Ranking Data with `rank()`

`rank()` assigns ranks to data values, useful for comparing relative standings.

```
df['Rank'] = df['Score'].rank()
print(df)
```

By default, ranks are average of ties.

Rank with method options:

- `'average'` (default): assigns average rank to ties.
- `'min'`: lowest rank in the group.
- `'max'`: highest rank in the group.
- `'first'`: ranks assigned in order they appear.

```
df['Rank_min'] = df['Score'].rank(method='min', ascending=False)
print(df)
```

5.3.6 Summary

- Use **`sort_values()`** to sort by columns, specifying ascending order and handling NaNs.
- Use **`sort_index()`** to sort by row or column labels.
- Use **`rank()`** to assign relative ranks, customizing how ties are treated.

These sorting and ranking tools help you better explore, report, and analyze your data.

5.4 Practical Examples: Feature Engineering

Feature engineering is the process of creating new variables from raw data to improve analysis or model performance. In pandas, this often involves transforming, combining, or encoding existing columns. In this section, we walk through practical techniques including **creating interaction terms**, **binning numeric features**, **encoding categorical variables**, and **deriving new informative columns**—all with a realistic example.

5.4.1 Sample Dataset

We'll use a small dataset of online shoppers to demonstrate feature engineering.

Full runnable code:

```python
import pandas as pd

data = {
    'User': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [23, 45, 35, 28, 62],
    'Gender': ['F', 'M', 'M', 'M', 'F'],
    'PurchaseAmount': [120.0, 250.0, 80.0, 300.0, 150.0],
    'Membership': ['Gold', 'Silver', 'Silver', 'Gold', 'Bronze']
}

df = pd.DataFrame(data)
print(df)
```

5.4.2 Creating Interaction Terms

Interaction terms combine two or more features to capture relationships.

```python
# Interaction between Age and PurchaseAmount
df['Age_x_Purchase'] = df['Age'] * df['PurchaseAmount']
```

This new column could be useful in modeling to capture how age influences spending behavior.

5.4.3 Binning Continuous Variables

Binning groups continuous values into discrete intervals. Useful for modeling or readability.

```python
# Bin Age into categories
df['AgeGroup'] = pd.cut(df['Age'], bins=[0, 30, 50, 100], labels=['Young', 'Middle-aged', 'Senior'])
```

This creates a categorical feature for age-related segmentation.

5.4.4 Encoding Categorical Data

Machine learning models require numeric inputs. Convert categorical columns using encoding.

a. One-Hot Encoding for Membership Level

```python
membership_dummies = pd.get_dummies(df['Membership'], prefix='Membership')
df = pd.concat([df, membership_dummies], axis=1)
```

b. Label Encoding for Gender

```
df['GenderEncoded'] = df['Gender'].map({'F': 0, 'M': 1})
```

> Note: One-hot encoding is preferable when categories are non-ordinal, to avoid implying rank.

5.4.5 Creating Derived Features

Use logic to create informative columns.

```
# High-value customer: Purchase over 200
df['HighValueCustomer'] = df['PurchaseAmount'] > 200
```

This binary column flags users for special targeting.

5.4.6 Final Engineered Dataset

```
print(df)
```

Output Preview:

```
      User  Age Gender  PurchaseAmount Membership  Age_x_Purchase    AgeGroup  \
0    Alice   23      F           120.0       Gold          2760.0       Young
1      Bob   45      M           250.0     Silver         11250.0  Middle-
aged
2 Charlie   35      M            80.0     Silver          2800.0  Middle-
aged
3    David   28      M           300.0       Gold          8400.0       Young
4      Eva   62      F           150.0     Bronze          9300.0      Senior

   Membership_Bronze  Membership_Gold  Membership_Silver  GenderEncoded  \
0                  0                1                  0              0
1                  0                0                  1              1
2                  0                0                  1              1
3                  0                1                  0              1
4                  1                0                  0              0

   HighValueCustomer
0              False
1               True
2              False
3               True
```

```
4               False
```

5.4.7 Summary

In this hands-on example, we engineered new features by:

- Creating interaction terms (`Age * PurchaseAmount`)
- Binning continuous values (`AgeGroup`)
- Encoding categorical data (`Membership`, `Gender`)
- Flagging binary conditions (`HighValueCustomer`)

These transformations enhance the dataset's analytical and predictive power. In real-world projects, well-engineered features can make the difference between a mediocre and an outstanding model. Practice with your own data to discover which features best represent your problem space.

Chapter 6.

Working with Text and Categorical Data

1. String Methods and Regular Expressions
2. Converting to and Working with Categorical Types
3. Handling Dates and Times with `pd.to_datetime()`
4. Practical Examples: Processing Textual and Time Series Data

6 Working with Text and Categorical Data

6.1 String Methods and Regular Expressions

Textual data is common in real-world datasets, especially when dealing with names, addresses, product descriptions, or logs. Pandas provides a rich set of **vectorized string operations** through the `.str` accessor, which allows you to apply string methods efficiently across entire Series objects.

These methods are crucial for **text cleaning**, **pattern extraction**, and **filtering**. Pandas string operations also integrate seamlessly with Python's built-in `re` (regular expression) module, enabling complex pattern matching.

6.1.1 Basic String Operations with .str

The `.str` accessor allows you to apply string functions element-wise on a Series. These are similar to Python's native string methods, but work on entire columns.

Full runnable code:

```python
import pandas as pd

data = {
    'Name': [' Alice ', 'BOB', 'charlie', 'dAvid ', 'EVA']
}

df = pd.DataFrame(data)

# Strip whitespace and convert to lowercase
df['Name_clean'] = df['Name'].str.strip().str.lower()
print(df)
```

Output:

```
      Name Name_clean
0   Alice      alice
1      BOB        bob
2  charlie    charlie
3   dAvid      david
4      EVA        eva
```

Other useful `.str` methods:

- `str.upper()` — Convert to uppercase
- `str.len()` — String length
- `str.startswith()`, `str.endswith()` — Filter based on prefixes/suffixes
- `str.replace()` — Replace substrings

- `str.contains()` — Check for substring or regex match

6.1.2 Filtering Text Data

Use `.str.contains()` to filter rows with matching substrings or patterns.

```
# Filter names that contain 'a' (case-insensitive)
filtered = df[df['Name_clean'].str.contains('a', case=False)]
print(filtered)
```

6.1.3 Regular Expressions with `re`

Pandas string methods support regex patterns by default. You can use regex to extract or replace complex patterns.

Extracting Digits from Strings

```
df2 = pd.DataFrame({
    'ProductID': ['AB-123', 'CD-456', 'EF-789']
})

# Extract numeric part
df2['Number'] = df2['ProductID'].str.extract(r'(\d+)', expand=False)
print(df2)
```

Output:

```
  ProductID Number
0    AB-123    123
1    CD-456    456
2    EF-789    789
```

Replacing Patterns

```
# Remove dashes using regex
df2['ProductID_clean'] = df2['ProductID'].str.replace(r'-', '', regex=True)
```

6.1.4 Multiple String Transformations

String methods can be chained for complex processing:

```
df3 = pd.DataFrame({
    'Comment': ['Great product!', '  poor service', 'Excellent.', 'AVERAGE!']
})
```

```
# Clean and categorize feedback
df3['Cleaned'] = df3['Comment'].str.strip().str.lower().str.replace(r'[^\w\s]', '', regex=True)
print(df3)
```

Output:

```
         Comment        Cleaned
0  Great product!  great product
1   poor service    poor service
2     Excellent.      excellent
3       AVERAGE!        average
```

6.1.5 Summary

Pandas string operations using `.str` give you fast, vectorized ways to manipulate and clean textual data. These methods are:

- Similar to Python string functions but optimized for Series.
- Fully compatible with regular expressions (`re`) for pattern matching and text extraction.
- Essential for preparing messy or unstructured text data for analysis or modeling.

By mastering `.str`, `str.contains()`, `str.extract()`, and regex patterns, you'll be well-equipped to process and analyze real-world textual data effectively.

6.2 Converting to and Working with Categorical Types

Categorical data is a pandas data type designed for fields with a **fixed number of possible values**—such as product categories, user roles, or geographic regions. These values are known as **categories**, and they can be either unordered or ordered.

Using categorical data types can significantly **reduce memory usage** and **speed up operations**, especially when performing filtering, sorting, or group-based aggregations.

6.2.1 Why Use Categorical Data?

- **Memory Efficiency**: Instead of storing repeated strings, pandas stores each unique value once and references it internally using integers.
- **Performance Boost**: Grouping, sorting, and comparisons are faster on categorical columns compared to `object` type.
- **Semantic Clarity**: Categorical types clearly signal that a field has a fixed set of

values.

6.2.2 Creating Categorical Columns

You can convert a column to categorical using `.astype('category')` or `pd.Categorical()`.

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'Department': ['Sales', 'HR', 'IT', 'HR', 'Sales', 'IT', 'IT']
})

# Convert to categorical
df['Department'] = df['Department'].astype('category')
print(df.dtypes)
```

Output:

```
Department    category
dtype: object
```

6.2.3 Memory Comparison

Let's compare memory usage before and after converting to categorical:

```
df_object = pd.DataFrame({'Dept': ['HR', 'Sales', 'IT'] * 1000})
df_category = df_object.copy()
df_category['Dept'] = df_category['Dept'].astype('category')

print(df_object.memory_usage(deep=True))
print(df_category.memory_usage(deep=True))
```

Typical result:

```
Dept    23000 bytes   # object type
Dept     3500 bytes   # category type
```

That's a **significant reduction** for large datasets.

6.2.4 Working with Categories

You can inspect and manage categories using `.cat` accessor:

```python
# View categories
print(df['Department'].cat.categories)

# Add a new category
df['Department'] = df['Department'].cat.add_categories(['Legal'])

# Remove a category
df['Department'] = df['Department'].cat.remove_categories(['Legal'])
```

6.2.5 Ordered Categories

Some categorical data has a natural order—like "Low", "Medium", "High". You can define this explicitly.

```python
priority = pd.Series(['High', 'Low', 'Medium', 'Low'])
priority_cat = priority.astype(pd.api.types.CategoricalDtype(
    categories=['Low', 'Medium', 'High'], ordered=True
))

# Now comparisons work as expected:
print(priority_cat > 'Low')
```

Output:

```
0     True
1    False
2     True
3    False
dtype: bool
```

6.2.6 Categorical in GroupBy

Grouping by a categorical column is faster than grouping by a string-based column.

```python
df = pd.DataFrame({
    'Department': ['HR', 'IT', 'Sales', 'IT', 'Sales'],
    'Salary': [50000, 60000, 55000, 62000, 57000]
})
df['Department'] = df['Department'].astype('category')

# Group by category
print(df.groupby('Department').mean(numeric_only=True))
```

6.2.7 Summary

Categorical data types in pandas:

- Reduce memory footprint for repetitive string fields.
- Improve performance in sorting and grouping.
- Allow ordering and controlled category management.

Use `.astype('category')` to convert columns and `.cat` accessor to fine-tune behavior. Categorical variables are not just about efficiency—they're a powerful tool for organizing your data semantically and computationally.

6.3 Handling Dates and Times with `pd.to_datetime()`

Date and time data is central to many data science tasks—from time series forecasting and trend analysis to log processing and event tracking. However, raw date strings often need to be parsed and standardized before meaningful analysis can occur. Pandas simplifies datetime handling with its powerful `pd.to_datetime()` function, which converts strings and other formats into standardized `datetime64[ns]` objects.

6.3.1 Why Datetime Processing Matters

Datetime data:

- Enables **time-based filtering**, sorting, and resampling.
- Supports **feature extraction** (year, month, weekday, etc.).
- Allows **time-based indexing** and calculations (e.g., durations, rolling windows).

Accurate and consistent datetime formatting is essential for reliability in all these operations.

6.3.2 Converting Strings to Datetime with `pd.to_datetime()`

Use `pd.to_datetime()` to convert a column of strings or integers into pandas datetime objects.

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'event': ['purchase', 'signup', 'login'],
    'date_str': ['2024-05-01', '2024/05/03', 'May 5, 2024']
})

df['date'] = pd.to_datetime(df['date_str'])
print(df)
```

Output:

```
     event      date_str       date
0  purchase    2024-05-01 2024-05-01
1    signup    2024/05/03 2024-05-03
2     login  May 5, 2024  2024-05-05
```

> `pd.to_datetime()` is flexible with formats and handles many common date patterns automatically.

6.3.3 Parsing Custom Date Formats

When the format is ambiguous or non-standard, you can specify it explicitly using the `format` argument for better performance and precision.

```
df = pd.DataFrame({
    'date_str': ['01-05-2024', '03-05-2024']
})

# European format: DD-MM-YYYY
df['date'] = pd.to_datetime(df['date_str'], format='%d-%m-%Y')
print(df)
```

6.3.4 Handling Missing or Invalid Dates

Invalid or missing values are handled gracefully using the `errors` parameter.

```
df = pd.DataFrame({
    'date_str': ['2024-05-01', 'invalid-date', None]
})

df['date'] = pd.to_datetime(df['date_str'], errors='coerce')
print(df)
```

Output:

```
     date_str       date
0  2024-05-01 2024-05-01
1  invalid-date       NaT
2          None       NaT
```

- `NaT` stands for "Not a Time", similar to `NaN` for missing numbers.

- Use `errors='coerce'` to convert bad entries to `NaT` safely.

6.3.5 Extracting Date Components

Once in datetime format, you can extract components using the `.dt` accessor:

```
df = pd.DataFrame({
    'timestamp': pd.to_datetime(['2024-01-15 14:30', '2024-03-22 09:45'])
})

df['year'] = df['timestamp'].dt.year
df['month'] = df['timestamp'].dt.month
df['day'] = df['timestamp'].dt.day
df['hour'] = df['timestamp'].dt.hour
df['weekday'] = df['timestamp'].dt.day_name()

print(df)
```

6.3.6 Datetime as Index

Datetime columns can be set as the index, enabling powerful time-based operations:

```
df.set_index('timestamp', inplace=True)
print(df.loc['2024-01'])  # Filter by month
```

6.3.7 Summary

- Use **`pd.to_datetime()`** to convert raw date strings into pandas datetime objects.
- Handle **custom formats**, **invalid values**, and **missing entries** using parameters like `format` and `errors`.
- Leverage `.dt` to **extract parts** of a datetime and `.set_index()` for **time-based indexing**.

Datetime handling is a core pandas skill—clean, parsed date fields open the door to insightful time-aware analysis.

6.4 Practical Examples: Processing Textual and Time Series Data

In this section, we'll walk through a practical example that combines **text cleaning**, **categorical encoding**, and **datetime parsing**—common tasks when analyzing real-world datasets such as logs, surveys, or time-stamped customer interactions.

Let's simulate analyzing a simple **customer service log** that includes time stamps, text-based categories, and message content.

6.4.1 Sample Dataset

```
import pandas as pd

data = {
    'timestamp': [
        '2024-06-01 09:15', '2024-06-01 10:45', '2024-06-01 14:20',
        '2024-06-02 08:30', '2024-06-02 12:10'
    ],
    'department': ['support', 'sales', 'SUPPORT', 'Sales', 'technical'],
    'message': [
        'Password reset requested.',
        'Inquiry about pricing plans.',
        'Account locked after login attempts.',
        'Discount code not working!',
        'Need help with device installation.'
    ]
}

df = pd.DataFrame(data)
```

6.4.2 Step 1: Parse Timestamps

Convert the `timestamp` column to datetime so we can extract temporal features.

```
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['date'] = df['timestamp'].dt.date
df['hour'] = df['timestamp'].dt.hour
```

Now we can group by date or analyze hourly activity.

6.4.3 Step 2: Clean and Standardize Text Data

The `department` column contains inconsistencies (e.g., capitalization). Normalize it using string methods:

```
df['department_clean'] = df['department'].str.strip().str.lower()
```

Now all departments are lowercase and comparable.

6.4.4 Step 3: Convert to Categorical

For memory and performance optimization, convert department to categorical:

```
df['department_cat'] = df['department_clean'].astype('category')
```

This improves groupby efficiency and adds semantic meaning to the data.

6.4.5 Step 4: Extract Keywords or Flags from Message Text

Suppose we want to identify whether a message involves login issues or pricing. Use `.str.contains()` with keyword filters:

```
df['is_login_issue'] = df['message'].str.contains('login|password|account', case=False)
df['is_pricing_related'] = df['message'].str.contains('pricing|discount', case=False)
```

6.4.6 Step 5: Aggregate Insights

Now we can generate summary statistics, such as how many login-related issues occurred per department:

```
summary = df.groupby('department_cat')[['is_login_issue', 'is_pricing_related']].sum()
print(summary)
```

6.4.7 Final DataFrame Preview

```
print(df[['timestamp', 'department', 'department_cat', 'hour', 'is_login_issue', 'is_pricing_related']]
```

Sample Output:

```
            timestamp department department_cat  hour  is_login_issue  is_pricing_relate
0 2024-06-01 09:15:00    support        support     9            True                Fals
```

```
1 2024-06-01 10:45:00      sales         sales    10          False              Tru
2 2024-06-01 14:20:00    SUPPORT       support    14           True             Fals
3 2024-06-02 08:30:00      Sales         sales     8          False              Tru
4 2024-06-02 12:10:00  technical     technical    12          False             Fals
```

6.4.8 Summary

In this example, we performed an end-to-end transformation of a time-stamped textual dataset:

- Parsed timestamps with `pd.to_datetime()`
- Cleaned and normalized text using `.str` methods
- Encoded standardized text with categorical types
- Extracted flags from messages using regex
- Aggregated and summarized behavior by department

These techniques are broadly applicable for real-world tasks like analyzing logs, support tickets, or survey responses. Mastering this combination of tools equips you to extract powerful insights from messy, mixed-type data.

Chapter 7.

Combining and Merging Datasets

1. Concatenation and Appending
2. Merge and Join: Inner, Outer, Left, Right Joins
3. Handling Duplicates and Conflicts
4. Practical Examples: Combining Multiple Data Sources

7 Combining and Merging Datasets

7.1 Concatenation and Appending

In many real-world scenarios, data is split across multiple files, time periods, or sources. **Combining** these fragments into a single DataFrame is a common task in data preparation. Pandas provides two main tools for this:

- `pd.concat()` for flexible vertical or horizontal concatenation.
- `.append()` (now deprecated) for simple row-wise addition.

Understanding how these methods align rows and columns—and how they handle mismatches—is key to effective dataset construction.

7.1.1 Concatenating DataFrames with `pd.concat()`

The `pd.concat()` function is the most general-purpose tool for combining pandas objects along a particular axis.

Vertical Concatenation (stacking rows)

Full runnable code:

```
import pandas as pd

# Monthly sales data
jan = pd.DataFrame({'Product': ['A', 'B'], 'Sales': [100, 150]})
feb = pd.DataFrame({'Product': ['A', 'B'], 'Sales': [120, 180]})

# Stack by rows (axis=0)
sales_q1 = pd.concat([jan, feb], axis=0, ignore_index=True)
print(sales_q1)
```

Output:

```
  Product  Sales
0       A    100
1       B    150
2       A    120
3       B    180
```

- `axis=0` means rows are stacked (default).
- `ignore_index=True` resets the index after concatenation.

Horizontal Concatenation (combining columns)

```
features_1 = pd.DataFrame({'ID': [1, 2], 'Height': [5.5, 6.0]})
features_2 = pd.DataFrame({'ID': [1, 2], 'Weight': [130, 155]})
```

```
# Merge by columns (axis=1)
combined = pd.concat([features_1, features_2.drop('ID', axis=1)], axis=1)
print(combined)
```

Output:

```
   ID  Height  Weight
0   1     5.5     130
1   2     6.0     155
```

7.1.2 Handling Mismatched Columns

If DataFrames have different columns, `pd.concat()` will **align columns by name** and fill missing entries with `NaN`.

```
mar = pd.DataFrame({'Product': ['A', 'B'], 'Revenue': [200, 250]})
stacked = pd.concat([jan, mar], axis=0, ignore_index=True)
print(stacked)
```

Output:

```
  Product  Sales  Revenue
0       A  100.0      NaN
1       B  150.0      NaN
2       A    NaN    200.0
3       B    NaN    250.0
```

7.1.3 Appending DataFrames with .append()

Pandas also provides the `.append()` method for quick row-wise additions. However, this method is **deprecated** in favor of `pd.concat()`.

```
# Append feb to jan
appended = jan.append(feb, ignore_index=True)
print(appended)
```

> YES **Note:** Use `pd.concat()` instead of `.append()` in new codebases for better performance and clarity.

7.1.4 Concatenation with Keys (Hierarchical Indexing)

You can use the `keys` argument to add multi-level index labels, which can be helpful when stacking datasets from different groups.

```
sales = pd.concat([jan, feb], keys=['Jan', 'Feb'])
print(sales)
```

Output:

```
        Product  Sales
Jan 0         A    100
    1         B    150
Feb 0         A    120
    1         B    180
```

Access data by key:

```
print(sales.loc['Jan'])
```

7.1.5 Summary

- Use **`pd.concat()`** to combine DataFrames **vertically** (`axis=0`) or **horizontally** (`axis=1`).
- Use **`ignore_index=True`** to reset indexes when stacking rows.
- Handle **mismatched columns** carefully—pandas fills missing values with `NaN`.
- The `.append()` method is deprecated; prefer `pd.concat()` for future-proof code.
- Use `keys` in `pd.concat()` for hierarchical indexing when working with labeled groups.

These techniques form the backbone of merging and stacking fragmented datasets into structured forms ready for analysis.

7.2 Merge and Join: Inner, Outer, Left, Right Joins

Merging datasets is a core operation in data analysis. In pandas, the `merge()` and `join()` functions allow you to combine datasets in ways similar to SQL joins. These operations are powerful for integrating data from multiple sources—such as combining customer records with their transaction history.

7.2.1 Overview of `merge()` and `join()`

- **`pd.merge()`** is the most flexible and widely used function for joining two DataFrames.

- `.join()` is a convenience method for joining on index (or optionally on columns).

Both allow for different **types of joins**:

- **Inner Join**: Keep only rows with keys present in both tables.
- **Left Join**: Keep all rows from the left table, fill in matches from the right.
- **Right Join**: Keep all rows from the right table, fill in matches from the left.
- **Outer Join**: Keep all rows from both tables, using `NaN` where no match is found.

7.2.2 Sample DataFrames

Let's create two example DataFrames for demonstration:

```
import pandas as pd

customers = pd.DataFrame({
    'customer_id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})

orders = pd.DataFrame({
    'order_id': [101, 102, 103, 104],
    'customer_id': [1, 2, 2, 4],
    'amount': [250, 150, 100, 300]
})
```

7.2.3 Inner Join

Keeps only matching `customer_id`s found in **both** DataFrames.

```
result = pd.merge(customers, orders, on='customer_id', how='inner')
print(result)
```

Output:

```
   customer_id     name  order_id  amount
0            1    Alice       101     250
1            2      Bob       102     150
2            2      Bob       103     100
```

Note: Customer `Charlie` (ID 3) and order with customer ID 4 are excluded.

7.2.4 Left Join

Keeps **all customers**, adding order info where available.

```
result = pd.merge(customers, orders, on='customer_id', how='left')
print(result)
```

Output:

```
   customer_id     name  order_id  amount
0            1    Alice     101.0   250.0
1            2      Bob     102.0   150.0
2            2      Bob     103.0   100.0
3            3  Charlie       NaN     NaN
```

Customer `Charlie` has no orders, so those fields are `NaN`.

7.2.5 Right Join

Keeps **all orders**, showing customer names if available.

```
result = pd.merge(customers, orders, on='customer_id', how='right')
print(result)
```

Output:

```
   customer_id     name  order_id  amount
0            1    Alice       101     250
1            2      Bob       102     150
2            2      Bob       103     100
3            4      NaN       104     300
```

Customer ID 4 has no matching entry in the customer table.

7.2.6 Outer Join

Combines all keys from both tables.

```
result = pd.merge(customers, orders, on='customer_id', how='outer')
print(result)
```

Output:

```
   customer_id     name  order_id  amount
0            1    Alice     101.0   250.0
1            2      Bob     102.0   150.0
2            2      Bob     103.0   100.0
3            3  Charlie       NaN     NaN
4            4      NaN     104.0   300.0
```

7.2.7 Handling Overlapping Column Names

If both DataFrames have columns with the same name other than the key, use `suffixes` to differentiate:

```
df1 = pd.DataFrame({'id': [1], 'score': [90]})
df2 = pd.DataFrame({'id': [1], 'score': [95]})

merged = pd.merge(df1, df2, on='id', suffixes=('_math', '_science'))
print(merged)
```

7.2.8 Merging on Multiple Keys

You can also merge using more than one column as the key.

```
sales = pd.DataFrame({
    'store': ['A', 'B', 'A'],
    'product': ['apple', 'banana', 'banana'],
    'revenue': [100, 200, 150]
})

targets = pd.DataFrame({
    'store': ['A', 'A', 'B'],
    'product': ['apple', 'banana', 'banana'],
    'target': [90, 140, 210]
})

merged = pd.merge(sales, targets, on=['store', 'product'], how='inner')
print(merged)
```

7.2.9 Using `.join()` for Index-Based Merging

`.join()` is ideal for joining on index:

```
left = pd.DataFrame({'value': [100, 200]}, index=['a', 'b'])
right = pd.DataFrame({'label': ['x', 'y']}, index=['a', 'c'])

joined = left.join(right, how='left')
print(joined)
```

7.2.10 Summary

- Use **`merge()`** to combine datasets using flexible keys and join types.
- Choose `how='inner'`, `'left'`, `'right'`, or `'outer'` based on the inclusion logic you need.

- Resolve **column name conflicts** with `suffixes`.
- Merge on **multiple keys** for more granular matching.
- Use `.join()` for quick joins on index.

By mastering `merge()` and `join()`, you'll be equipped to combine relational datasets cleanly and efficiently—an essential skill in data wrangling and analysis.

7.3 Handling Duplicates and Conflicts

When combining datasets from multiple sources, issues such as **duplicate rows** and **conflicting information** are common. These problems can lead to incorrect aggregations, inflated row counts, or misleading analysis. Pandas provides tools like `duplicated()`, `drop_duplicates()`, and merge parameters such as `indicator=True` to help detect and resolve these issues.

7.3.1 Detecting Duplicates

The `duplicated()` method helps identify rows that are duplicates of previous ones.

```python
import pandas as pd

df = pd.DataFrame({
    'id': [1, 2, 2, 3, 4, 4],
    'name': ['Alice', 'Bob', 'Bob', 'Charlie', 'Dan', 'Dan']
})

# Detect duplicates
print(df.duplicated())
```

Output:

```
0    False
1    False
2     True
3    False
4    False
5     True
dtype: bool
```

Rows 2 and 5 are marked as duplicates of earlier rows.

7.3.2 Removing Duplicates

Use `drop_duplicates()` to remove them:

```
df_cleaned = df.drop_duplicates()
print(df_cleaned)
```

Output:

```
   id     name
0   1    Alice
1   2      Bob
3   3  Charlie
4   4      Dan
```

You can also specify **subset columns** to consider only certain fields:

```
df_unique_names = df.drop_duplicates(subset='name')
```

7.3.3 Conflicting Data After Merge

Joining datasets may result in conflicting information for the same key. For example:

```
customers = pd.DataFrame({
    'customer_id': [1, 2],
    'email': ['alice@example.com', 'bob@example.com']
})

survey = pd.DataFrame({
    'customer_id': [2, 3],
    'email': ['bobby@newmail.com', 'charlie@example.com']
})

merged = pd.merge(customers, survey, on='customer_id', how='outer', suffixes=('_left', '_right'))
print(merged)
```

Output:

```
   customer_id         email_left          email_right
0            1  alice@example.com                  NaN
1            2    bob@example.com    bobby@newmail.com
2            3                NaN  charlie@example.com
```

Here, customer ID 2 has two conflicting email addresses.

7.3.4 Tracking Merge Source with `indicatorTrue`

Add a special column to track which DataFrame contributed each row.

```
merged_with_flag = pd.merge(customers, survey, on='customer_id', how='outer', indicator=True)
print(merged_with_flag)
```

Output:

```
   customer_id              email_x              email_y      _merge
0            1    alice@example.com                  NaN   left_only
1            2      bob@example.com    bobby@newmail.com        both
2            3                  NaN  charlie@example.com  right_only
```

The `_merge` column helps you analyze and resolve discrepancies—like preferring one source over another or manually validating overlaps.

7.3.5 Manual Conflict Resolution

When you encounter conflicting values, you may choose one over the other, or create a rule:

```
# Prefer survey email if available
merged['email_final'] = merged['email_right'].combine_first(merged['email_left'])
```

This keeps the right-side (newer) email if it exists; otherwise, it uses the left.

7.3.6 Summary

- Use **`duplicated()`** to detect and **`drop_duplicates()`** to remove redundant rows.
- Specify **`subset`** to control which columns to check.
- When merging, handle **column name conflicts** using `suffixes=('_left', '_right')`.
- Use **`indicator=True`** to track row origin and manage overlaps.
- Apply **manual rules** or `.combine_first()` to resolve conflicting data based on business logic.

By detecting and cleaning duplicates and resolving conflicts carefully, you ensure your merged datasets remain accurate, consistent, and ready for analysis.

7.4 Practical Examples: Combining Multiple Data Sources

In real-world data science projects, datasets often come from various sources—such as CSV exports, web APIs, or databases—and must be **cleaned and combined** before analysis. This section walks through a practical example: combining customer, orders, and product datasets to create a unified view for sales analysis.

7.4.1 Step 1: Load Multiple Datasets

We'll simulate three different sources:

```python
import pandas as pd

# Simulated CSV import (customer data)
customers = pd.DataFrame({
    'customer_id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie'],
    'email': ['alice@mail.com', 'bob@mail.com', 'charlie@mail.com']
})

# Simulated API response (orders)
orders = pd.DataFrame({
    'order_id': [101, 102, 103, 104],
    'customer_id': [1, 2, 2, 4],  # Note: customer_id 4 is not in customers
    'product_id': [1001, 1002, 1001, 1003],
    'quantity': [2, 1, 3, 1]
})

# Simulated local CSV (products data)
products = pd.DataFrame({
    'product_id': [1001, 1002, 1003],
    'product_name': ['Laptop', 'Phone', 'Tablet'],
    'price': [1200, 800, 500]
})
```

7.4.2 Step 2: Clean and Prepare Each Dataset

Check for missing values and duplicates:

```python
# Drop duplicates if any
customers.drop_duplicates(inplace=True)
orders.drop_duplicates(inplace=True)
products.drop_duplicates(inplace=True)

# Ensure IDs are of correct type
orders['customer_id'] = orders['customer_id'].astype('Int64')
```

7.4.3 Step 3: Merge Datasets

Start by merging orders with customer information:

```python
orders_customers = pd.merge(
    orders, customers, on='customer_id', how='left', indicator=True
)
print(orders_customers)
```

Note that customer ID 4 (in orders) has no match in the customers table.

You can filter or flag these cases:

```
# Show orders with missing customer data
missing_customers = orders_customers[orders_customers['_merge'] == 'left_only']
```

Now merge with product data:

```
full_data = pd.merge(
    orders_customers.drop(columns=['_merge']),
    products,
    on='product_id',
    how='left'
)
```

7.4.4 Step 4: Create Computed Columns

Generate a new column for order value:

```
full_data['total'] = full_data['quantity'] * full_data['price']
```

7.4.5 Step 5: Final DataFrame Preview

```
print(full_data[['order_id', 'name', 'product_name', 'quantity', 'total']])
```

Output:

```
   order_id     name product_name  quantity   total
0       101    Alice       Laptop         2  2400.0
1       102      Bob        Phone         1   800.0
2       103      Bob       Laptop         3  3600.0
3       104      NaN       Tablet         1   500.0
```

The last row shows a missing customer (ID 4), which could be filtered, flagged, or logged.

7.4.6 Step 6: Handle Errors and Ensure Reproducibility

Wrap logic in a reusable function with error handling:

```
def load_and_combine(customers, orders, products):
    try:
        df = pd.merge(orders, customers, on='customer_id', how='left')
        df = pd.merge(df, products, on='product_id', how='left')
        df['total'] = df['quantity'] * df['price']
        return df
    except Exception as e:
        print("Error during merge:", e)
        return pd.DataFrame()
```

7.4.7 Summary

In this end-to-end example, we:

- Loaded datasets from multiple "sources"
- Cleaned duplicates and checked for key mismatches
- Used `merge()` to combine related data using common keys
- Calculated new columns for business insight
- Demonstrated best practices like error handling and reproducibility

This pattern—load, clean, combine, compute—is the foundation for any scalable, analysis-ready data pipeline.

Chapter 8.

Grouping and Aggregation

1. GroupBy: Splitting, Applying, and Combining
2. Aggregation Functions and Custom Aggregations
3. Pivot Tables and Cross-tabulations
4. Practical Examples: Summarizing and Analyzing Data

8 Grouping and Aggregation

8.1 GroupBy: Splitting, Applying, and Combining

One of the most powerful features in pandas is the **GroupBy** mechanism, which follows a **“split–apply–combine”** strategy:

- **Split** the data into groups based on one or more keys (e.g., a column or index level).
- **Apply** a function to each group independently (e.g., sum, mean, transformation).
- **Combine** the results back into a DataFrame or Series.

This approach is ideal for summarizing, transforming, or filtering data based on categories such as region, time, or product.

8.1.1 Basic GroupBy Syntax

To group a DataFrame, use the `.groupby()` method:

```python
import pandas as pd

# Example dataset: sales data
data = pd.DataFrame({
    'region': ['North', 'South', 'North', 'East', 'South', 'East'],
    'category': ['A', 'A', 'B', 'B', 'A', 'B'],
    'sales': [100, 200, 150, 120, 300, 250]
})
```

8.1.2 Grouping by a Single Column

```python
grouped = data.groupby('region')
print(grouped['sales'].sum())
```

Output:

```
region
East     370
North    250
South    500
Name: sales, dtype: int64
```

This groups the data by `region` and sums the `sales` for each.

8.1.3 Grouping by Multiple Columns

```
result = data.groupby(['region', 'category'])['sales'].sum()
print(result)
```

Output:

```
region  category
East    B           370
North   A           100
        B           150
South   A           500
Name: sales, dtype: int64
```

Grouping by both `region` and `category` provides a multi-index Series showing the sales breakdown at a finer granularity.

8.1.4 Grouping by Index Level

You can also group using index levels in multi-indexed DataFrames:

```
data2 = data.set_index(['region', 'category'])
print(data2.groupby(level=0).sum())
```

This groups by the first level of the index (`region`) and aggregates the rest.

8.1.5 Aggregation vs Transformation vs Filtering

Aggregation produces a reduced version of the data using summary statistics.

```
data.groupby('category')['sales'].agg(['sum', 'mean', 'count'])
```

Transformation returns a DataFrame the same shape as the original, with values transformed within each group. Common use: normalization.

```
# Normalize sales within each region
data['sales_norm'] = data.groupby('region')['sales'].transform(lambda x: x / x.sum())
```

Filtering returns a subset of the original data where a condition on the group is met.

```
# Keep only regions where total sales > 300
filtered = data.groupby('region').filter(lambda x: x['sales'].sum() > 300)
print(filtered)
```

8.1.6 Applying Custom Functions

You can apply any function, including user-defined functions, to each group using `.apply()`:

```
def max_minus_min(series):
    return series.max() - series.min()

result = data.groupby('category')['sales'].apply(max_minus_min)
print(result)
```

8.1.7 Resetting Index After Grouping

To return grouped results as a flat DataFrame:

```
flat = data.groupby(['region', 'category'], as_index=False)['sales'].sum()
print(flat)
```

Output:

```
  region category  sales
0   East        B    370
1  North        A    100
2  North        B    150
3  South        A    500
```

8.1.8 Summary

The GroupBy pattern in pandas allows you to:

- Group by one or more keys or index levels.
- Perform **aggregations** (like `sum`, `mean`), **transformations** (e.g., normalization), or **filters**.
- Use `.agg()`, `.transform()`, and `.filter()` depending on the output shape and intent.

This functionality enables clear, concise analysis of structured datasets—whether you're computing total sales by region or comparing product performance by category.

8.2 Aggregation Functions and Custom Aggregations

Aggregating data is central to summarizing large datasets. After grouping data with `groupby()`, pandas allows you to compute summary statistics using built-in aggregation functions like `sum()`, `mean()`, and `count()`, or define custom functions using lambdas or named Python functions.

These operations help answer questions such as:

- What is the total sales per region?
- What is the average order size by customer?
- How many products fall into each category?

8.2.1 Built-in Aggregation Functions

After grouping, you can apply a single aggregation method:

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'region': ['North', 'South', 'North', 'East', 'South', 'East'],
    'sales': [100, 200, 150, 120, 300, 250],
    'orders': [1, 2, 1, 1, 3, 2]
})

# Total sales per region
print(df.groupby('region')['sales'].sum())
```

Output:

```
region
East     370
North    250
South    500
Name: sales, dtype: int64
```

Common built-in aggregation functions include:

- `.sum()` – Total
- `.mean()` – Average
- `.count()` – Number of non-null entries
- `.min()` / `.max()` – Minimum or maximum

8.2.2 Multiple Aggregations with agg()

The `.agg()` method allows applying multiple aggregations at once:

```
summary = df.groupby('region')['sales'].agg(['sum', 'mean', 'max'])
print(summary)
```

Output:

```
        sum   mean  max
region
East     370  185.0   250
North    250  125.0   150
South    500  250.0   300
```

You can also use different aggregations for different columns:

```
summary = df.groupby('region').agg({
    'sales': 'sum',
    'orders': 'mean'
})
print(summary)
```

8.2.3 Custom Aggregation Functions

You're not limited to built-in methods—you can define custom aggregators using lambda or named functions.

Example 1: Lambda Function

```
# Range of sales per region
sales_range = df.groupby('region')['sales'].agg(lambda x: x.max() - x.min())
print(sales_range)
```

Output:

```
region
East     130
North     50
South    100
Name: sales, dtype: int64
```

Example 2: Named Function

```
def coefficient_of_variation(x):
    return x.std() / x.mean()

variation = df.groupby('region')['sales'].agg(coefficient_of_variation)
print(variation)
```

This measures the relative variability in sales per region.

8.2.4 Naming Aggregations

To label your output clearly, use tuples in `.agg()`:

```python
summary = df.groupby('region')['sales'].agg([
    ('Total Sales', 'sum'),
    ('Average', 'mean'),
    ('High Sale', 'max')
])
print(summary)
```

Output:

```
        Total Sales  Average  High Sale
region
East            370   185.00        250
North           250   125.00        150
South           500   250.00        300
```

8.2.5 Summary

- Use built-in aggregations (`sum()`, `mean()`, etc.) to quickly summarize grouped data.
- Apply multiple aggregations simultaneously using `.agg()`.
- Define **custom aggregation functions** for advanced summaries.
- Combine groupby and aggregation to derive insights efficiently and concisely.

Whether you're analyzing regional sales, customer activity, or inventory performance, aggregation is a foundational tool for producing actionable summaries in pandas.

8.3 Pivot Tables and Cross-tabulations

In addition to `groupby()`, pandas provides two powerful tools for summarizing and analyzing data in tabular formats: **pivot tables** using `pivot_table()` and **cross-tabulations** using `pd.crosstab()`. These are especially useful for multidimensional aggregation, such as comparing metrics across multiple categories or generating frequency tables.

8.3.1 Pivot Tables with `pivot_table()`

The `pivot_table()` function creates a new table by reorganizing and aggregating data. It's similar to Excel pivot tables and provides a concise way to summarize datasets.

Basic Syntax

```python
pd.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean')
```

- `data`: the DataFrame
- `values`: column(s) to aggregate
- `index`: rows in the pivot table
- `columns`: columns in the pivot table
- `aggfunc`: aggregation function (default is `'mean'`, others include `'sum'`, `'count'`, etc.)

Example: Sales by Region and Product

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'region': ['North', 'South', 'North', 'East', 'South', 'East'],
    'product': ['A', 'A', 'B', 'B', 'A', 'B'],
    'sales': [100, 200, 150, 120, 300, 250]
})

pivot = pd.pivot_table(df, values='sales', index='region', columns='product', aggfunc='sum', fill_value
print(pivot)
```

Output:

```
product    A    B
region
East       0  370
North    100  150
South    500    0
```

This shows total sales by region and product. Missing combinations are filled with 0 using `fill_value`.

8.3.2 Multiple Aggregations in Pivot Tables

You can pass a list of aggregation functions:

```
multi_agg = pd.pivot_table(df, values='sales', index='region', columns='product',
                           aggfunc=['sum', 'mean'], fill_value=0)
print(multi_agg)
```

This produces a multi-level column index with both sum and mean for each product.

8.3.3 Pivot Table vs `groupby()`

Feature	`groupby()`	`pivot_table()`
Output	Series or DataFrame	Always a DataFrame
Multi-dim Support	Needs chaining/grouping	Built-in via `index` and `columns` args
Multiple aggfuncs	Supported via `.agg()`	Supported via `aggfunc`
Fill missing data	Manual	Easy with `fill_value`

8.3.4 Cross-tabulations with `pd.crosstab()`

`pd.crosstab()` is used for computing **frequency tables**, especially useful for categorical variables.

Example: Frequency of Products Sold by Region

```
freq = pd.crosstab(index=df['region'], columns=df['product'])
print(freq)
```

Output:

```
product  A  B
region
East     0  2
North    1  1
South    2  0
```

This counts the number of times each product appears in each region.

Adding Margins

You can include row and column totals with `margins=True`:

```
freq_totals = pd.crosstab(df['region'], df['product'], margins=True)
print(freq_totals)
```

Output:

```
product  A  B  All
region
East     0  2    2
North    1  1    2
South    2  0    2
All      3  3    6
```

8.3.5 Summary

- Use `pivot_table()` to summarize and reshape data across multiple dimensions with aggregation.
- Set `index` and `columns` to define the structure of the table, and use `aggfunc` for summary logic.
- Prefer `pivot_table()` over `groupby()` for 2D summaries.
- Use `pd.crosstab()` for counting occurrences in categorical combinations.
- These tools are invaluable for building summary reports, dashboards, and quick data analysis.

By mastering pivot tables and cross-tabulations, you'll be equipped to extract deep insights from multi-category datasets in a compact, readable format.

8.4 Practical Examples: Summarizing and Analyzing Data

In this section, we'll walk through a practical scenario where **grouping, aggregation, and pivoting** help uncover key business insights. The example demonstrates how a company might analyze **monthly revenue by product and region**, a common task in sales and operations reporting.

8.4.1 Scenario: Monthly Sales Revenue Analysis

Let's say you work with a dataset of e-commerce transactions, and you want to analyze **total revenue** per month, broken down by **product category** and **region**.

Step 1: Prepare the Sample Dataset

```
import pandas as pd

# Sample sales data
data = pd.DataFrame({
    'order_id': [1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008],
    'order_date': [
        '2024-01-15', '2024-01-18', '2024-02-02', '2024-02-25',
        '2024-03-01', '2024-03-10', '2024-03-15', '2024-03-20'
    ],
    'region': ['North', 'South', 'North', 'South', 'East', 'North', 'South', 'East'],
    'category': ['Electronics', 'Clothing', 'Electronics', 'Clothing',
                 'Clothing', 'Electronics', 'Clothing', 'Clothing'],
    'revenue': [1200, 500, 1500, 600, 450, 1800, 550, 400]
})

# Convert order_date to datetime
data['order_date'] = pd.to_datetime(data['order_date'])
```

```
# Extract the month (as period for grouping)
data['month'] = data['order_date'].dt.to_period('M')
```

8.4.2 Step 2: Grouping and Aggregating Total Revenue

Group data by `month` and `region` to see how sales vary across time and geography:

```
monthly_region_sales = data.groupby(['month', 'region'])['revenue'].sum().reset_index()
print(monthly_region_sales)
```

Output:

```
     month region  revenue
0  2024-01  North     1200
1  2024-01  South      500
2  2024-02  North     1500
3  2024-02  South      600
4  2024-03   East      850
5  2024-03  North     1800
6  2024-03  South      550
```

8.4.3 Step 3: Pivoting for Better Visualization

Pivot the grouped data so each region becomes a column and months are rows:

```
pivot = monthly_region_sales.pivot(index='month', columns='region', values='revenue').fillna(0)
print(pivot)
```

Output:

```
region    East  North  South
month
2024-01    0.0  1200.0  500.0
2024-02    0.0  1500.0  600.0
2024-03  850.0  1800.0  550.0
```

Now you can quickly compare how revenue changes across months and regions. For example:

- North had steady growth from January to March.
- East only had revenue in March.
- South peaked in February.

8.4.4 Step 4: Further Analysis by Category

To analyze revenue by product category per month:

```
monthly_category_sales = data.groupby(['month', 'category'])['revenue'].sum().unstack()
print(monthly_category_sales)
```

Output:

```
category  Clothing  Electronics
month
2024-01      500.0       1200.0
2024-02      600.0       1500.0
2024-03     1400.0       1800.0
```

This shows:

- Electronics consistently outperformed clothing.
- March had the highest sales in both categories.

8.4.5 Step 5: Insights and Takeaways

From this simple dataset, we extracted useful insights:

- Revenue increased month-over-month across all regions and categories.
- Electronics sales dominated in every month, but clothing sales surged in March.
- The East region contributed only in March, suggesting a possible new market expansion or delayed launch.

Such grouped summaries are not only powerful for dashboards and reports, but also for feeding into models (e.g., forecasting, segmentation).

8.4.6 Summary

In this example, we:

- Extracted temporal features (`month`) from datetime data.
- Grouped by multiple keys (`month`, `region`, `category`).
- Aggregated revenue using `.sum()`.
- Pivoted results for clearer visualization.

By combining pandas' `groupby()`, `agg()`, and `pivot_table()` methods, you can quickly move from raw data to meaningful summaries and actionable insights—an essential skill in any data science workflow.

Chapter 9.

Time Series and Date Functionality

9 Time Series and Date Functionality

9.1 Working with DateTimeIndex and PeriodIndex

Time series data—data indexed by time—is foundational in domains like finance, sensor analytics, and forecasting. Pandas provides two specialized index types for working with time: **`DateTimeIndex`** and **`PeriodIndex`**. These allow efficient time-based slicing, filtering, and frequency-aware calculations.

9.1.1 Converting to Datetime with `pd.to_datetime()`

Most time series data comes with timestamp columns as strings. You must convert them to pandas datetime objects using `pd.to_datetime()`:

Full runnable code:

```
import pandas as pd

df = pd.DataFrame({
    'date': ['2024-01-01', '2024-01-02', '2024-01-03'],
    'price': [100, 101, 99]
})

# Convert to datetime
df['date'] = pd.to_datetime(df['date'])
print(df.dtypes)
```

Output:

```
date     datetime64[ns]
price             int64
dtype: object
```

9.1.2 Setting `DateTimeIndex`

Once the column is in datetime format, set it as the index to enable time-aware operations:

```
df = df.set_index('date')
print(df)
```

Output:

```
            price
date
2024-01-01    100
```

```
2024-01-02    101
2024-01-03     99
```

Now you can filter or slice using date strings:

```
# Get all data from January 2nd onwards
print(df['2024-01-02':])
```

9.1.3 Understanding `DateTimeIndex`

When the index is datetime-based, pandas enables powerful features:

- **Time-based slicing** (`df['2024-01']`)
- **Resampling** and **rolling** operations
- **Attribute access** like `index.year`, `index.month`, etc.

You can also directly create a `DateTimeIndex`:

```
dates = pd.date_range(start='2024-01-01', periods=5, freq='D')
print(dates)
```

9.1.4 Working with `PeriodIndex`

While `DateTimeIndex` tracks exact timestamps, `PeriodIndex` is for fixed-duration intervals like months, quarters, or years—great for accounting or financial periods.

```
# Create a period-based index
periods = pd.period_range(start='2024-01', periods=3, freq='M')
sales = pd.Series([1000, 1200, 1100], index=periods)
print(sales)
```

Output:

```
2024-01    1000
2024-02    1200
2024-03    1100
Freq: M, dtype: int64
```

This representation is more semantically accurate when working with interval-based data, like monthly reports.

You can convert a `datetime` column to a `PeriodIndex`:

```
df['period'] = df.index.to_period('M')
print(df.head())
```

9.1.5 Frequency Attribute

Both `DateTimeIndex` and `PeriodIndex` support the `.freq` attribute, which indicates the time frequency (`'D'`, `'M'`, `'Q'`, etc.). You can set or infer it:

```
# Ensure the index has a defined frequency
df = df.asfreq('D')
```

This is helpful before resampling or interpolating.

9.1.6 Practical Example: Stock Prices

```
stock_data = pd.DataFrame({
    'date': pd.date_range(start='2024-01-01', periods=6, freq='D'),
    'price': [100, 102, 101, 105, 104, 106]
})
stock_data = stock_data.set_index('date')

# Slice by time
print(stock_data['2024-01-03':'2024-01-05'])
```

Output:

```
              price
date
2024-01-03      101
2024-01-04      105
2024-01-05      104
```

9.1.7 Summary

- Use `pd.to_datetime()` to convert strings to timestamps.
- Set the datetime column as the index to enable powerful time-based operations.
- Use `DateTimeIndex` for exact timestamps; use `PeriodIndex` for interval-based time (like months or quarters).
- Use `.asfreq()` or `date_range()` to define or enforce frequency.

These time-aware index types form the backbone of pandas' time series functionality, making it easier to analyze trends, compare periods, and manipulate temporal data efficiently.

9.2 Resampling and Frequency Conversion

Resampling is a key feature of pandas for time series data, allowing you to change the frequency of observations — for instance, converting **daily stock prices to monthly averages** or **hourly sensor data to 15-minute intervals**. This process can be used for **downsampling** (reducing frequency) or **upsampling** (increasing frequency), and is commonly paired with aggregation or interpolation.

Pandas provides the `.resample()` method to handle this efficiently when working with a `DateTimeIndex` or `PeriodIndex`.

9.2.1 Downsampling: Aggregating to a Lower Frequency

Downsampling reduces the time frequency by summarizing data. For example, you can compute the monthly average from daily values.

```
import pandas as pd

# Create sample daily stock prices
data = pd.DataFrame({
    'date': pd.date_range(start='2024-01-01', periods=15, freq='D'),
    'price': [100, 101, 99, 102, 104, 105, 107, 108, 110, 112, 115, 117, 116, 114, 113]
})

data.set_index('date', inplace=True)

# Resample to monthly frequency and take the mean
monthly_avg = data.resample('M').mean()
print(monthly_avg)
```

Output:

```
              price
date
2024-01-31  108.133
```

Here, `'M'` denotes month-end frequency. Other common frequencies include `'D'` (daily), `'W'` (weekly), `'Q'` (quarterly), etc.

9.2.2 Upsampling: Increasing Frequency

Upsampling increases the frequency by inserting new timestamps, which usually requires filling or interpolating missing values.

```
# Upsample to 12-hour frequency
upsampled = data.resample('12H').asfreq()
print(upsampled.head())
```

This introduces `NaN` values for new timestamps:

```
             price
date
2024-01-01   100.0
2024-01-01 12:00   NaN
2024-01-02   101.0
2024-01-02 12:00   NaN
2024-01-03    99.0
```

To fill these gaps, you can use methods like forward-fill (`ffill`) or interpolation:

```
# Forward fill missing values
filled = upsampled.ffill()

# Or interpolate linearly
interpolated = upsampled.interpolate(method='time')
```

9.2.3 Common Aggregation Functions in Resampling

You can specify aggregation functions such as `'mean'`, `'sum'`, `'max'`, `'count'`, etc.

```
# Weekly max price
weekly_max = data.resample('W').max()
print(weekly_max)
```

9.2.4 Example: Sensor Data Every 10 Minutes

```
sensor_data = pd.DataFrame({
    'timestamp': pd.date_range(start='2024-01-01 00:00', periods=6, freq='10T'),
    'temperature': [22.1, 22.3, 22.2, 22.5, 22.4, 22.6]
})
sensor_data.set_index('timestamp', inplace=True)

# Resample to 30-minute intervals using mean
resampled = sensor_data.resample('30T').mean()
print(resampled)
```

Output:

```
                     temperature
timestamp
2024-01-01 00:00        22.200
2024-01-01 00:30        22.450
2024-01-01 01:00        22.600
```

This shows how to aggregate high-frequency measurements for a more compact view.

9.2.5 Summary

- Use `.resample()` to change the frequency of time series data.
- **Downsampling** aggregates to lower frequency (e.g., daily → monthly).
- **Upsampling** increases frequency and often requires filling or interpolating missing values.
- Combine `.resample()` with aggregation (`mean()`, `sum()`, etc.) or filling (`ffill()`, `interpolate()`).
- Ideal for summarizing stock data, weather logs, sensor output, and more.

With resampling, you can align time series data to the scale most appropriate for your analysis or reporting needs.

9.3 Rolling and Expanding Windows

Time series data often exhibits trends, noise, or seasonality. To analyze such patterns, **moving window operations** are invaluable. Pandas provides two powerful tools for this:

- `rolling()`: Applies calculations over a sliding window of fixed size.
- `expanding()`: Grows the window from the start of the data to the current point.

These are commonly used for **smoothing**, **trend analysis**, and **cumulative statistics** in financial, environmental, and sensor data.

9.3.1 Using `rolling()`: Moving Window Calculations

The `rolling()` method computes statistics over a fixed-size window that moves incrementally along the time axis.

Full runnable code:

```
import pandas as pd

# Sample time series data (daily stock prices)
data = pd.DataFrame({
    'date': pd.date_range(start='2024-01-01', periods=10, freq='D'),
    'price': [100, 102, 101, 105, 107, 110, 111, 108, 109, 112]
})
data.set_index('date', inplace=True)

# 3-day moving average
data['rolling_mean'] = data['price'].rolling(window=3).mean()
print(data)
```

Output:

```
            price  rolling_mean
date
2024-01-01    100           NaN
2024-01-02    102           NaN
2024-01-03    101    101.000000
2024-01-04    105    102.666667
2024-01-05    107    104.333333
...
```

The first two rows are `NaN` because the window size (3) isn't yet satisfied. You can control this behavior using `min_periods=1`.

```
data['rolling_sum'] = data['price'].rolling(window=3, min_periods=1).sum()
```

9.3.2 Custom Functions with `rolling()`

You can apply custom functions using `.apply()`:

```
# Rolling range (max - min) over 3 days
data['rolling_range'] = data['price'].rolling(window=3).apply(lambda x: x.max() - x.min())
```

This helps capture volatility or deviation within the window.

9.3.3 Using `expanding()`: Cumulative Statistics

Unlike `rolling()`, `expanding()` starts from the first data point and expands the window as it moves forward.

```
# Cumulative average
data['expanding_mean'] = data['price'].expanding().mean()
```

Output:

```
            price  expanding_mean
date
2024-01-01    100      100.000000
2024-01-02    102      101.000000
2024-01-03    101      101.000000
2024-01-04    105      102.000000
...
```

This is useful for long-term trend analysis or monitoring average behavior over time.

9.3.4 Example: Sensor Data Rolling Analysis

```
sensor = pd.DataFrame({
    'timestamp': pd.date_range(start='2024-01-01', periods=6, freq='H'),
    'temperature': [22.1, 22.4, 22.2, 22.5, 22.8, 23.0]
})
sensor.set_index('timestamp', inplace=True)

# 3-hour moving average
sensor['temp_avg_3h'] = sensor['temperature'].rolling(window=3).mean()

# Expanding max temperature
sensor['expanding_max'] = sensor['temperature'].expanding().max()
print(sensor)
```

9.3.5 Summary

- `rolling(window=N)` applies functions like `mean()` or `sum()` over a fixed-size window.
- `expanding()` performs cumulative operations from the start of the data.
- Both accept custom functions via `.apply()`.
- Use `min_periods` to control NaN behavior in partial windows.
- Ideal for computing moving averages, volatility, and cumulative metrics in time series.

These tools enhance your ability to detect trends and smooth noisy signals, making them essential in time series analysis with pandas.

9.4 Practical Examples: Financial and Sensor Data

In this section, we'll walk through a practical time series workflow using **financial stock prices** and **sensor data**. You'll see how to leverage pandas for indexing by time, resampling data, applying rolling averages, and preparing for visualization—all with interpretation and awareness of potential pitfalls.

9.4.1 Example 1: Financial Stock Data Analysis

Let's simulate stock price data and analyze its trends using pandas time series functionality.

Full runnable code:

```
import pandas as pd
import numpy as np
```

```
# Simulated stock price data (daily)
dates = pd.date_range(start="2024-01-01", periods=20, freq="D")
prices = [100 + np.random.randn() for _ in range(20)]

df = pd.DataFrame({'date': dates, 'price': prices})
df.set_index('date', inplace=True)

print(df.head())
```

Resampling: Weekly Average Price

To smooth fluctuations and understand weekly behavior, we resample the data to a weekly frequency:

```
weekly_avg = df.resample('W').mean()
print(weekly_avg)
```

This shows the average closing price per week. **Pitfall:** If your data has missing dates (e.g., weekends), resampling without handling gaps may skew results. Ensure your index is continuous or account for market holidays.

Rolling Window: 5-Day Moving Average

A rolling average smooths short-term noise and highlights trends:

```
df['rolling_5d'] = df['price'].rolling(window=5).mean()
print(df[['price', 'rolling_5d']].tail())
```

Interpretation: If the 5-day average is rising while daily prices are volatile, the trend is upward. **Pitfall:** The first few rows will be `NaN` due to insufficient data in the window.

Preparing for Plotting

To prepare this data for visualization (e.g., using matplotlib or seaborn), structure it clearly:

```
df.reset_index().plot(x='date', y=['price', 'rolling_5d'], title="Stock Price and 5-Day Moving Average"
```

9.4.2 Example 2: IoT Sensor Data

Let's now simulate hourly temperature readings from a sensor.

```
sensor_data = pd.DataFrame({
    'timestamp': pd.date_range(start='2024-01-01 00:00', periods=48, freq='H'),
    'temperature': np.random.normal(loc=22, scale=1.0, size=48)
})
sensor_data.set_index('timestamp', inplace=True)
```

Resampling to 6-Hour Intervals

```
resampled = sensor_data.resample('6H').mean()
print(resampled.head())
```

This can reveal broader temperature trends and reduce noise.

Rolling Temperature Change

Calculate a 3-hour rolling temperature change:

```
sensor_data['temp_3h_mean'] = sensor_data['temperature'].rolling(window=3).mean()
```

Use case: Detect unusual heating or cooling patterns (e.g., HVAC malfunction).

9.4.3 Common Pitfalls

- **Missing Timestamps:** Be aware that `resample()` creates regular intervals. If your original data is sparse, interpolating or forward-filling may be necessary.
- **Time Zone Handling:** Convert to consistent time zones using `.tz_convert()` if working with data from multiple sources.
- **NaNs in Rolling Windows:** Early rows may have `NaN` values. Handle them explicitly with `min_periods` or fill strategies (`.fillna()`).

9.4.4 Summary

This example highlights how pandas empowers time series analysis:

- Convert and set datetime indexes for slicing and filtering.
- Use `resample()` to aggregate at new frequencies.
- Apply `rolling()` for trend detection and smoothing.
- Prepare structured data for visual storytelling.

Mastering these workflows is essential for real-world projects involving financial markets, IoT sensors, and beyond.

Chapter 10.

Input and Output with Pandas

1. Reading and Writing CSV, Excel, JSON, SQL
2. Handling Large Datasets: Chunking and Efficient I/O
3. Working with HDF5 and Parquet Files
4. Practical Examples: Data Persistence and Exchange

10 Input and Output with Pandas

10.1 Reading and Writing CSV, Excel, JSON, SQL

Pandas makes it straightforward to read from and write to various data file formats and databases, allowing seamless integration with your data workflow. This section covers essential I/O functions: how to load and save **CSV**, **Excel**, **JSON**, and **SQL** data, along with common parameters and pitfalls.

10.1.1 Reading and Writing CSV Files

CSV (Comma-Separated Values) is one of the most common data formats.

Reading a CSV:

```
import pandas as pd

# Load CSV file
df = pd.read_csv('data/sales.csv', delimiter=',', encoding='utf-8', header=0, index_col=0)
print(df.head())
```

- `delimiter` (or `sep`): Specifies the column separator (default is `,`).
- `encoding`: Useful when working with non-ASCII characters (e.g., `'utf-8'`).
- `header`: Row number(s) to use as the column names (default is 0).
- `index_col`: Column(s) to use as the row index.

Writing a CSV:

```
df.to_csv('data/cleaned_sales.csv', index=True, encoding='utf-8')
```

- `index`: Whether to write row labels (index) to the file.
- Always verify if the index should be saved to avoid duplicates or confusion later.

10.1.2 Reading and Writing Excel Files

Excel files (`.xlsx`, `.xls`) are popular in business.

Reading Excel:

```
# Read Excel sheet
excel_df = pd.read_excel('data/report.xlsx', sheet_name='Sheet1', index_col=0)
print(excel_df.head())
```

- `sheet_name`: Name or number of the sheet to read.
- `index_col`: Column to use as index.
- Pandas uses `openpyxl` or `xlrd` internally (ensure packages are installed).

Writing Excel:

```
excel_df.to_excel('data/output_report.xlsx', sheet_name='Summary', index=False)
```

- You can write multiple DataFrames to different sheets using `ExcelWriter`.

10.1.3 Reading and Writing JSON Files

JSON is common for nested and hierarchical data.

Reading JSON:

```
json_df = pd.read_json('data/data.json', encoding='utf-8')
print(json_df.head())
```

- `orient` parameter controls the expected JSON format (`'records'`, `'split'`, etc.).
- JSON often requires preprocessing if deeply nested.

Writing JSON:

```
json_df.to_json('data/output.json', orient='records', lines=True)
```

- `orient='records'` outputs a list of JSON objects.
- `lines=True` writes JSON objects per line (useful for streaming).

10.1.4 Reading and Writing SQL Databases

Pandas can read from and write to SQL databases, facilitating integration with persistent data storage.

Reading from SQL:

```
import sqlite3

# Connect to SQLite database
conn = sqlite3.connect('data/database.db')

# Read SQL query into DataFrame
sql_df = pd.read_sql('SELECT * FROM sales_data WHERE amount > 100', conn)
print(sql_df.head())
```

- You can use any SQLAlchemy-compatible database.
- The query can be a raw SQL string or SQLAlchemy expression.

Writing to SQL:

```
# Write DataFrame to SQL table (replace if exists)
df.to_sql('cleaned_sales', conn, if_exists='replace', index=False)
```

- `if_exists` can be `'fail'`, `'replace'`, or `'append'`.

- Be careful with the schema and data types when writing.

10.1.5 Common Pitfalls and Tips

- **Missing Data:** When reading, missing values might appear as `NaN`. Use `na_values` to specify additional representations.

```
pd.read_csv('data.csv', na_values=['NA', 'missing'])
```

- **Data Types:** Pandas infers data types, but sometimes conversion is needed after loading.

```
df['date'] = pd.to_datetime(df['date'])
df['quantity'] = df['quantity'].astype(int)
```

- **Index Handling:** When saving, be explicit about saving indexes to avoid duplications or misalignments on reload.
- **Encoding Issues:** Always specify the correct encoding, especially with non-English data, to prevent errors.

10.1.6 Summary

Format	Read Function	Write Function	Key Parameters
CSV	`read_csv()`	`to_csv()`	`delimiter`, `encoding`, `index_col`
Excel	`read_excel()`	`to_excel()`	`sheet_name`, `index_col`
JSON	`read_json()`	`to_json()`	`orient`, `lines`
SQL	`read_sql()`	`to_sql()`	`if_exists`, `index`, connection

With these core I/O functions, you can efficiently move data between pandas and various external sources, enabling smooth data ingestion and export workflows.

10.2 Handling Large Datasets: Chunking and Efficient I/O

When working with large datasets, loading an entire file into memory at once can be impractical or even impossible due to memory constraints. Pandas provides several strategies to efficiently handle large data files, enabling you to process data incrementally or optimize resource usage.

10.2.1 The Challenge of Large Files

Large files — often gigabytes in size — can cause your system to run out of memory if loaded fully. This can lead to crashes or extremely slow performance. To manage this, we use **chunking**, which reads data in smaller pieces, allowing for incremental processing.

10.2.2 Reading Files in Chunks with `chunksize`

The `chunksize` parameter in `read_csv()` (and some other readers) lets you read a file piece by piece as an iterator of DataFrames:

```python
import pandas as pd

chunk_iter = pd.read_csv('large_data.csv', chunksize=100000)

for i, chunk in enumerate(chunk_iter):
    print(f"Processing chunk {i+1}")
    # Example operation: calculate sum of a column in the chunk
    total_sales = chunk['sales'].sum()
    print(f"Total sales in chunk {i+1}: {total_sales}")
```

- Each `chunk` is a DataFrame with `chunksize` rows.
- You can process each chunk independently, then aggregate results to avoid loading the full file.

10.2.3 Aggregating Results Across Chunks

Suppose you want the overall total sales without loading the entire dataset:

```python
total_sales = 0
for chunk in pd.read_csv('large_data.csv', chunksize=100000):
    total_sales += chunk['sales'].sum()

print(f"Overall total sales: {total_sales}")
```

This technique lets you efficiently summarize or filter large data.

10.2.4 Streaming and Memory Management

- **Streaming:** Using `chunksize` creates a lazy iterator, reading only part of the file at a time.
- **Memory:** After processing a chunk, Python's garbage collector frees that memory, avoiding overload.
- **Selective columns:** Use the `usecols` parameter to read only necessary columns,

reducing memory usage.

```
chunk_iter = pd.read_csv('large_data.csv', chunksize=100000, usecols=['date', 'sales'])
```

10.2.5 Writing Data Efficiently in Chunks

When processing and writing large data in pieces, use the `mode` and `header` parameters in `to_csv()` to append chunks to a file without overwriting:

```
for i, chunk in enumerate(pd.read_csv('large_data.csv', chunksize=100000)):
    processed_chunk = chunk[chunk['sales'] > 1000]  # filter example
    if i == 0:
        processed_chunk.to_csv('filtered_sales.csv', index=False, mode='w', header=True)
    else:
        processed_chunk.to_csv('filtered_sales.csv', index=False, mode='a', header=False)
```

This approach allows you to create a filtered or transformed dataset incrementally.

10.2.6 Optimizing Reading and Writing

- **Specify data types:** Use the `dtype` parameter to reduce memory by assigning efficient types upfront.

  ```
  dtypes = {'sales': 'float32', 'product_id': 'int32'}
  chunk_iter = pd.read_csv('large_data.csv', chunksize=100000, dtype=dtypes)
  ```

- **Compression:** Read/write compressed files (`.gz`, `.zip`) by setting `compression` parameter, saving disk space and I/O time.

10.2.7 Summary

- Large files can be processed efficiently using **chunking** with `chunksize`.
- Process each chunk independently, aggregating or filtering as needed.
- Control memory use with selective columns (`usecols`), specifying `dtype`, and compressing files.
- Writing in chunks supports incremental data transformations and filtered output.
- These techniques make Pandas suitable for big data workflows without needing specialized tools.

Mastering chunking and efficient I/O ensures your data analysis scales gracefully with data volume and system resources.

10.3 Working with HDF5 and Parquet Files

When working with large datasets or requiring fast disk I/O and efficient storage, binary file formats like **HDF5** and **Parquet** offer significant advantages over plain text formats like CSV or JSON. These formats provide compression, faster read/write speeds, and better integration with big data ecosystems.

10.3.1 HDF5 Format

HDF5 (Hierarchical Data Format version 5) is a versatile, high-performance binary format designed for storing large amounts of numerical data. It supports complex data hierarchies and is widely used in scientific computing.

Pandas provides built-in support for HDF5 through the `read_hdf()` and `to_hdf()` functions.

Writing a DataFrame to HDF5:

```
import pandas as pd

df = pd.DataFrame({
    'date': pd.date_range('2024-01-01', periods=5),
    'value': [10, 20, 30, 40, 50]
})

# Write to HDF5 file under the key 'data'
df.to_hdf('data_store.h5', key='data', mode='w')
```

Reading from HDF5:

```
df_loaded = pd.read_hdf('data_store.h5', key='data')
print(df_loaded)
```

- **`key`** specifies the group name inside the HDF5 file, allowing multiple datasets in one file.
- HDF5 supports compression and fast random access.
- It works well for numeric or tabular data and supports querying subsets.

10.3.2 Parquet Format

Parquet is a columnar storage file format optimized for analytics workloads. It offers efficient compression and encoding schemes, making it a go-to format in big data tools like Apache Spark, Hadoop, and cloud platforms.

Pandas supports Parquet files with `read_parquet()` and `to_parquet()` functions, but requires either `pyarrow` or `fastparquet` as a dependency.

Writing a DataFrame to Parquet:

```
df.to_parquet('data.parquet', index=False)
```

Reading from Parquet:

```
df_parquet = pd.read_parquet('data.parquet')
print(df_parquet)
```

- Parquet files store data by column, enabling faster reads for column-based queries.
- The format supports schema evolution, nested data, and is interoperable across many systems.
- Common in machine learning pipelines and cloud data lakes.

10.3.3 When to Use HDF5 vs Parquet?

Feature	HDF5	Parquet
Data model	Hierarchical, flexible	Columnar, optimized for tables
Use case	Scientific computing, complex datasets	Big data analytics, cloud storage
Compression	Yes	Yes
Supported by	Pandas, HDF5 libraries	Pandas, Apache Arrow ecosystem
Performance	Fast for complex queries	Fast columnar queries
Interoperability	Limited outside scientific tools	Broad, supports many platforms

10.3.4 Practical Example: Combining Formats

```
# Create example DataFrame
df = pd.DataFrame({
    'id': [1, 2, 3, 4],
    'category': ['A', 'B', 'A', 'B'],
    'value': [10, 20, 30, 40]
})

# Save to HDF5
df.to_hdf('example.h5', key='df', mode='w')

# Save to Parquet
df.to_parquet('example.parquet')

# Load both
df_hdf = pd.read_hdf('example.h5', 'df')
df_parquet = pd.read_parquet('example.parquet')

print("From HDF5:\n", df_hdf)
print("\nFrom Parquet:\n", df_parquet)
```

10.3.5 Summary

- **HDF5** is great for hierarchical datasets and works well in scientific environments needing flexible data storage.
- **Parquet** shines in analytics, offering fast, compressed columnar storage compatible with big data tools.
- Both formats greatly improve performance and reduce storage compared to CSV or JSON.
- Pandas' `read_hdf()`, `to_hdf()`, `read_parquet()`, and `to_parquet()` make working with these formats easy and efficient.

Adopting these binary formats can significantly enhance your data pipeline's speed and scalability, especially for large-scale or production-grade workflows.

10.4 Practical Examples: Data Persistence and Exchange

In real-world data science projects, you often need to load data from multiple sources, clean or transform it, then save it in formats suitable for sharing, reporting, or future analysis. This section walks through a practical example demonstrating these steps using Pandas, highlighting best practices for choosing file formats and ensuring data integrity.

10.4.1 Scenario: Combining Sales and Customer Data

Suppose you have two data sources:

- A **CSV file** with monthly sales transactions.
- An **Excel file** with customer details.

Your goal is to merge these datasets, perform some analysis, then save the cleaned data in formats that optimize future access and sharing.

10.4.2 Step 1: Read Data from Multiple Sources

```python
import pandas as pd

# Read sales data from CSV
sales_df = pd.read_csv('sales_jan.csv')

# Read customer data from Excel (first sheet)
customers_df = pd.read_excel('customers.xlsx', sheet_name='Sheet1')
```

```
print(sales_df.head())
print(customers_df.head())
```

10.4.3 Step 2: Data Cleaning and Merging

Let's ensure consistency, handle missing values, and merge the datasets on `customer_id`.

```
# Fill missing sales amounts with 0
sales_df['amount'] = sales_df['amount'].fillna(0)

# Standardize customer IDs (ensure same type)
sales_df['customer_id'] = sales_df['customer_id'].astype(str)
customers_df['customer_id'] = customers_df['customer_id'].astype(str)

# Merge sales with customer info (left join to keep all sales)
merged_df = pd.merge(sales_df, customers_df, on='customer_id', how='left')

print(merged_df.info())
```

10.4.4 Step 3: Analyze and Add Features

Add a new feature: categorize sales into 'High' and 'Low' based on amount.

```
merged_df['sales_category'] = merged_df['amount'].apply(lambda x: 'High' if x > 500 else 'Low')
```

10.4.5 Step 4: Save Processed Data for Different Use Cases

- **CSV:** Simple sharing or quick inspection.
- **Parquet:** Efficient storage and future analysis with fast loading.
- **Excel:** For business users who prefer spreadsheets.

```
# Save to CSV (without index for clean file)
merged_df.to_csv('processed_sales.csv', index=False)

# Save to Parquet (efficient binary format)
merged_df.to_parquet('processed_sales.parquet', index=False)

# Save to Excel (multiple sheets possible if needed)
merged_df.to_excel('processed_sales.xlsx', index=False, sheet_name='SalesData')
```

10.4.6 Practical Tips for Format Choice and Data Integrity

- **CSV** is widely compatible but lacks support for data types and compression. Avoid for very large datasets.
- **Parquet** is ideal for performance and storing data types but requires compatible tools for reading.
- **Excel** is convenient for reporting but can be slower and less suited for very large data.
- Always check and **standardize data types** before merging to avoid subtle bugs.
- Use `index=False` to avoid saving unnecessary indexes unless explicitly needed.
- Consider **compression options** (`compression='gzip'` in CSV or Parquet) to save disk space.

10.4.7 Summary

This example demonstrated how to:

- Load data from CSV and Excel files.
- Clean and merge datasets using Pandas.
- Add meaningful features.
- Save data in multiple formats tailored to different use cases.

By mastering data persistence and exchange techniques, you ensure smooth collaboration, efficient workflows, and reproducible data science projects.

Chapter 11.

Visualization with Pandas

1. Basic Plotting with Pandas `.plot()`
2. Customizing Plots: Titles, Labels, and Styles
3. Integrating with Matplotlib and Seaborn
4. Practical Examples: Exploratory Data Analysis

11 Visualization with Pandas

11.1 Basic Plotting with Pandas `.plot()`

Pandas comes with built-in plotting capabilities that leverage the powerful Matplotlib library underneath. The `.plot()` method is available directly on Series and DataFrames, providing a convenient way to visualize data without switching contexts. This allows you to quickly explore your data's distributions, relationships, and trends.

11.1.1 Common Plot Types in Pandas

You can create a variety of plot types with `.plot()` by specifying the `kind` parameter. Here are some of the most common:

- **Line plot** (`kind='line'`): Default for Series and DataFrames, ideal for time series or ordered data.
- **Bar plot** (`kind='bar'` or `barh` for horizontal): Useful for categorical comparisons.
- **Histogram** (`kind='hist'`): Shows distribution of a numeric variable.
- **Box plot** (`kind='box'`): Visualizes the distribution and identifies outliers.
- **Scatter plot** (`kind='scatter'`): Shows relationships between two numeric variables (DataFrame only).

11.1.2 Basic Plotting Examples

Let's explore these plot types using a small example dataset.

Full runnable code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Sample DataFrame
data = pd.DataFrame({
    'Month': pd.date_range('2023-01-01', periods=6, freq='M'),
    'Sales': [250, 300, 280, 350, 400, 420],
    'Expenses': [200, 220, 210, 250, 270, 300],
    'Category': ['A', 'B', 'A', 'B', 'A', 'B']
})

# Set Month as index for plotting
data.set_index('Month', inplace=True)
```

1. Line Plot (Default)

```
data[['Sales', 'Expenses']].plot()
plt.title('Sales and Expenses Over Time')
plt.xlabel('Month')
plt.ylabel('Amount')
plt.show()
```

This line plot shows how sales and expenses vary across months, helping to identify trends.

2. Bar Plot

```
data.groupby('Category')['Sales'].sum().plot(kind='bar', color='skyblue', figsize=(6,4))
plt.title('Total Sales by Category')
plt.xlabel('Category')
plt.ylabel('Sales')
plt.show()
```

Bar plots summarize categorical data—in this case, total sales per category.

3. Histogram

```
data['Sales'].plot(kind='hist', bins=5, color='orange', alpha=0.7)
plt.title('Sales Distribution')
plt.xlabel('Sales Amount')
plt.show()
```

Histograms reveal the distribution of sales amounts, useful for spotting skewness or clusters.

4. Box Plot

```
data[['Sales', 'Expenses']].plot(kind='box')
plt.title('Sales and Expenses Distribution')
plt.show()
```

Box plots display quartiles and highlight potential outliers in the data.

5. Scatter Plot

Scatter plots require specifying `x` and `y` columns explicitly on a DataFrame:

```
data.plot(kind='scatter', x='Expenses', y='Sales', color='green')
plt.title('Sales vs Expenses')
plt.show()
```

This helps explore the relationship between expenses and sales.

11.1.3 Customizing Your Plots

The `.plot()` method accepts several useful parameters:

- **`color`**: Set the color of the plot elements.
- **`figsize`**: Specify the size of the figure in inches, e.g., `(8, 5)`.
- **`title`**: Add titles via Matplotlib's `plt.title()`.
- **`alpha`**: Control transparency, helpful for overlapping data.

Example combining parameters:

```
data['Sales'].plot(kind='line', color='purple', figsize=(10,4), alpha=0.8)
plt.title('Sales Over Time')
plt.show()
```

11.1.4 Summary

- Pandas `.plot()` offers quick, convenient visualization methods built on Matplotlib.
- Choose plot types based on your data: line for trends, bar for categories, histograms and boxplots for distributions, scatter for relationships.
- Customize plots with colors, figure sizes, and transparency to enhance readability.
- These simple visualizations are a great first step toward understanding your data before more complex analysis or styling.

With these tools, you can rapidly generate insightful charts directly from your data frames and series—making exploration fast and interactive.

11.2 Customizing Plots: Titles, Labels, and Styles

While pandas' `.plot()` method allows for quick visualizations, customizing your plots is crucial for making them presentation-ready and easier to interpret. You can enhance your plots by adding titles, axis labels, legends, and gridlines, as well as adjusting styles such as colors, line types, and markers. These customizations use a combination of `.plot()` parameters and Matplotlib functions (`matplotlib.pyplot`).

11.2.1 Basic Plot Customization

Let's start with a basic line plot using a sample dataset:

```
import pandas as pd
import matplotlib.pyplot as plt

data = pd.DataFrame({
    'Month': pd.date_range('2023-01-01', periods=6, freq='M'),
    'Revenue': [2500, 2700, 2600, 2900, 3100, 3300],
    'Profit': [500, 600, 550, 650, 700, 750]
})
data.set_index('Month', inplace=True)

# Basic line plot
data.plot()
plt.show()
```

11.2.2 Adding Titles and Axis Labels

To make plots informative, add titles and axis labels using Matplotlib functions:

```
ax = data.plot(figsize=(8, 4))
plt.title('Monthly Revenue and Profit (2023)', fontsize=14)
plt.xlabel('Month')
plt.ylabel('Amount ($)')
plt.grid(True)
plt.legend(title='Metrics')
plt.tight_layout()
plt.show()
```

- `plt.title()` adds a main title.
- `plt.xlabel()` and `plt.ylabel()` set axis labels.
- `plt.grid(True)` adds gridlines for readability.
- `plt.legend(title=...)` customizes the legend.

11.2.3 Changing Colors, Line Styles, and Markers

Pandas allows color, line style, and marker control via the `plot()` method:

```
data.plot(
    figsize=(8, 4),
    color=['#1f77b4', '#ff7f0e'],
    linestyle='--',
    marker='o'
)
plt.title('Revenue vs Profit')
plt.grid(True)
plt.show()
```

- `color`: Accepts a list of color codes or names.
- `linestyle`: `'--'`, `'-'`, `':'`, etc.
- `marker`: `'o'` for circle, `'s'` for square, `'x'` for cross, etc.

You can mix these in custom line settings by plotting columns individually:

```
ax = data['Revenue'].plot(color='green', linestyle='-', marker='s', label='Revenue')
data['Profit'].plot(ax=ax, color='blue', linestyle='--', marker='o', label='Profit')
plt.title('Styled Revenue and Profit Lines')
plt.legend()
plt.show()
```

11.2.4 Using Styles and Themes

Matplotlib supports built-in styles to change the overall look:

```
plt.style.use('ggplot')  # Options: 'seaborn', 'fivethirtyeight', 'classic', etc.
data.plot()
plt.title('Styled with ggplot')
plt.show()
```

Try different styles to match the tone of your presentation or report.

11.2.5 Saving Plots to Files

Use `plt.savefig()` to export your customized plots:

```
ax = data.plot()
plt.title('Monthly Business Performance')
plt.savefig('monthly_performance.png', dpi=300, bbox_inches='tight')
```

- `dpi=300` ensures high-resolution output.
- `bbox_inches='tight'` trims whitespace.

You can save in different formats: `.png`, `.jpg`, `.svg`, `.pdf`.

11.2.6 Summary

Customizing plots improves their effectiveness and readability. Here are key takeaways:

- Use `plt.title()`, `xlabel()`, and `ylabel()` for annotations.
- Style with `color`, `linestyle`, and `marker` parameters.
- Use `plt.style.use()` to switch themes.
- Always `savefig()` for reusable or report-ready graphics.

With these tools, you can transform basic plots into clear, compelling visualizations suited for both exploratory analysis and polished presentations.

11.3 Integrating with Matplotlib and Seaborn

While pandas' built-in `.plot()` method provides a fast and convenient way to visualize data, integrating with **Matplotlib** and **Seaborn** unlocks more powerful and flexible visualization tools. These libraries are particularly useful when you need customized layouts, complex plots, or statistical visualizations beyond what `.plot()` can offer.

11.3.1 Extending Pandas Plots with Matplotlib

Every pandas plot returns a **Matplotlib Axes object**, which allows further customization using Matplotlib commands. This is useful when you need precise control over plot elements like annotations, multiple subplots, or layout adjustments.

Example: Subplots with Matplotlib Axes

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame({
    'Month': pd.date_range('2023-01-01', periods=6, freq='M'),
    'Revenue': [2500, 2700, 2600, 2900, 3100, 3300],
    'Profit': [500, 600, 550, 650, 700, 750]
})
df.set_index('Month', inplace=True)

fig, axes = plt.subplots(2, 1, figsize=(8, 6), sharex=True)

df['Revenue'].plot(ax=axes[0], color='blue', title='Monthly Revenue')
df['Profit'].plot(ax=axes[1], color='green', title='Monthly Profit')

axes[0].set_ylabel('Revenue ($)')
axes[1].set_ylabel('Profit ($)')
plt.tight_layout()
plt.show()
```

11.3.2 Introducing Seaborn for Statistical Plots

Seaborn is built on top of Matplotlib and integrates well with pandas. It offers attractive default themes and statistical visualizations like:

- **Heatmaps** for correlation or frequency tables
- **Pairplots** to explore relationships between multiple variables
- **Regression plots** for linear trend analysis

To use Seaborn, you must import it and optionally set a theme:

```
import seaborn as sns
sns.set_theme(style="whitegrid")
```

11.3.3 Example: Heatmap from Correlation Matrix

```
import seaborn as sns
import matplotlib.pyplot as plt

# Correlation heatmap
corr = df.corr()
```

```
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

Heatmaps are great for identifying strong relationships or redundancies between numeric variables.

11.3.4 Example: Pairplot to Explore Relationships

```
# Add a category for illustration
df['Region'] = ['East', 'West', 'East', 'West', 'East', 'West']

sns.pairplot(df, hue='Region')
plt.suptitle('Pairwise Plots by Region', y=1.02)
plt.show()
```

Pairplots show scatter plots and histograms for each variable combination, with optional grouping via `hue`.

11.3.5 Example: Regression Plot

```
sns.regplot(x='Revenue', y='Profit', data=df, marker='o')
plt.title('Revenue vs. Profit Regression')
plt.show()
```

Seaborn automatically fits and plots a regression line with confidence intervals—helpful for detecting trends.

11.3.6 When to Use Pandas, Matplotlib, or Seaborn

Task	Preferred Tool
Quick exploration	`pandas.plot()`
Multi-axes/subplots	`matplotlib.pyplot`
Statistical or complex plots	`seaborn`
Full layout and figure control	`matplotlib.pyplot`
Publication-quality visuals	`seaborn` + `matplotlib`

11.3.7 Summary

- **Matplotlib** extends `.plot()` with full control over plot structure, annotations, and layout.
- **Seaborn** adds expressive statistical plots that look great by default.
- You can manipulate pandas data and seamlessly pass it into Seaborn for visualization.

By combining pandas for data wrangling and Seaborn/Matplotlib for visualization, you gain a robust and professional workflow suitable for both analysis and reporting.

11.4 Practical Examples: Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a critical step in any data science project. It helps you understand your data's structure, detect anomalies, test hypotheses, and uncover relationships using summaries and visualizations. In this section, we'll walk through a practical EDA workflow using a real-world dataset: a fictional retail sales dataset.

11.4.1 Step 1: Load and Inspect the Data

```python
import pandas as pd

# Simulated retail sales data
data = pd.DataFrame({
    'OrderDate': pd.date_range('2023-01-01', periods=12, freq='M'),
    'Region': ['North', 'South', 'East', 'West'] * 3,
    'Category': ['Electronics', 'Furniture', 'Office Supplies'] * 4,
    'Sales': [2000, 1800, 1500, 2200, 1700, 1600, 2100, 1900, 1550, 2300, 1750, 1650],
    'Profit': [300, 200, 100, 400, 150, 120, 350, 180, 130, 420, 160, 140]
})
data['OrderDate'] = pd.to_datetime(data['OrderDate'])
data.set_index('OrderDate', inplace=True)
data.head()
```

11.4.2 Step 2: Time Series Trend Analysis

Let's analyze monthly sales and profit trends.

```python
import matplotlib.pyplot as plt

data[['Sales', 'Profit']].plot(figsize=(10, 5), marker='o', title='Monthly Sales and Profit')
plt.ylabel('Amount ($)')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Interpretation: This line plot reveals fluctuations over time. You may notice sales peak mid-year while profits vary independently, indicating possible changes in margins or costs.

11.4.3 Step 3: Compare Categories

Group and compare total sales by product category.

```
category_sales = data.groupby('Category')['Sales'].sum().sort_values()

category_sales.plot(kind='barh', color='skyblue', title='Total Sales by Category')
plt.xlabel('Total Sales')
plt.tight_layout()
plt.show()
```

Interpretation: Bar plots are great for comparing categories. Here, you might find that Electronics lead in revenue, but it's also useful to compare with profits to see which category is more efficient.

11.4.4 Step 4: Regional Profit Comparison

```
region_profit = data.groupby('Region')['Profit'].mean()

region_profit.plot(kind='bar', color='orange', title='Average Profit by Region')
plt.ylabel('Avg Profit')
plt.tight_layout()
plt.show()
```

Interpretation: This comparison may reveal which regions are most profitable, helping guide strategic decisions such as investment or marketing.

11.4.5 Step 5: Correlation Analysis

We use a heatmap to examine relationships between numerical features.

```
import seaborn as sns

corr = data[['Sales', 'Profit']].corr()
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
```

Interpretation: A strong positive correlation between Sales and Profit would suggest that as sales rise, so does profit. Weak correlation might prompt deeper analysis.

11.4.6 Step 6: Multi-Factor Analysis

We can use pivot tables to examine average sales by both region and category.

```
pivot = pd.pivot_table(data, values='Sales', index='Region', columns='Category', aggfunc='mean')
pivot.plot(kind='bar', figsize=(10, 5), title='Avg Sales by Region and Category')
plt.ylabel('Avg Sales')
plt.tight_layout()
plt.show()
```

Interpretation: This visualization helps identify which category performs best in each region—valuable for targeted marketing.

11.4.7 Summary

In this example, we:

- Explored time trends with line plots
- Compared categories and regions using bar charts
- Investigated relationships with heatmaps
- Used pivot tables for multi-dimensional insights

This end-to-end EDA workflow combines pandas for data wrangling and Matplotlib/Seaborn for visualization, enabling clear and actionable insights. Effective EDA isn't just about code—it's about using visual storytelling to understand and communicate your data.

Chapter 12.

Performance Optimization and Best Practices

12 Performance Optimization and Best Practices

12.1 Memory Usage and DataFrame Size Reduction

As datasets grow, optimizing memory usage becomes essential for performance, especially on resource-constrained systems or when working with millions of rows. Pandas provides tools to inspect memory usage and offers several strategies to reduce memory footprint efficiently without losing data integrity.

12.1.1 Inspecting Memory Usage

The first step is to analyze how much memory your DataFrame consumes. You can use `.info()` for a general overview and `.memory_usage(deep=True)` for a detailed, column-level breakdown:

```
import pandas as pd
import numpy as np

# Sample dataset
df = pd.DataFrame({
    'ID': range(1, 10001),
    'Category': np.random.choice(['A', 'B', 'C', 'D'], size=10000),
    'Amount': np.random.randint(100, 1000, size=10000),
    'Discount': np.random.rand(10000)
})

print(df.info())
print(df.memory_usage(deep=True))
```

This will show that string-based columns (like `Category`) and 64-bit numeric types consume substantial memory.

12.1.2 Strategy 1: Convert to Categorical Types

For columns with repeated string values (like labels or categories), converting them to the `category` type can drastically reduce memory usage:

```
df['Category'] = df['Category'].astype('category')
```

This internally maps strings to integer codes, which are far more efficient.

12.1.3 Strategy 2: Use Smaller Numeric Types

By default, pandas uses 64-bit integers and floats (`int64`, `float64`). But if your data doesn't require that level of precision, downcasting can help:

```
df['ID'] = pd.to_numeric(df['ID'], downcast='unsigned')
df['Amount'] = pd.to_numeric(df['Amount'], downcast='integer')
df['Discount'] = pd.to_numeric(df['Discount'], downcast='float')
```

You can also use `df.dtypes` to inspect which columns may be good candidates for downcasting.

12.1.4 Strategy 3: Drop Unnecessary Columns

Columns like unique identifiers, metadata, or redundant features can often be dropped before processing:

```
df = df.drop(columns=['ID'])  # if not required for analysis
```

Removing even a single high-cardinality or text-heavy column can have a significant impact.

12.1.5 Strategy 4: Efficient Indexing

Default integer indexing is fine for most cases, but when working with time series or categorical data, setting an appropriate index can make joins, lookups, and resampling faster and more memory-efficient:

```
df = df.set_index('Category')  # only if Category is unique or frequently queried
```

Be cautious—setting indexes on high-cardinality columns (e.g., unique IDs) may actually increase memory usage.

12.1.6 Before and After: Memory Comparison

Let's compare the memory footprint before and after optimization:

```
df_orig = pd.DataFrame({
    'ID': range(1, 10001),
    'Category': np.random.choice(['A', 'B', 'C', 'D'], size=10000),
    'Amount': np.random.randint(100, 1000, size=10000),
    'Discount': np.random.rand(10000)
})

print("Original size (MB):", df_orig.memory_usage(deep=True).sum() / 1024 ** 2)

# Optimized version
```

```
df_opt = df_orig.copy()
df_opt['Category'] = df_opt['Category'].astype('category')
df_opt['ID'] = pd.to_numeric(df_opt['ID'], downcast='unsigned')
df_opt['Amount'] = pd.to_numeric(df_opt['Amount'], downcast='integer')
df_opt['Discount'] = pd.to_numeric(df_opt['Discount'], downcast='float')

print("Optimized size (MB):", df_opt.memory_usage(deep=True).sum() / 1024 ** 2)
```

Result: You'll typically see memory reductions of 40–70% depending on the data composition.

12.1.7 Summary

Optimizing memory usage in pandas is a simple yet powerful way to boost performance. Key techniques include:

- Inspecting memory with `.memory_usage(deep=True)`
- Using `category` for repetitive string data
- Downcasting numeric types to smaller representations
- Dropping irrelevant columns
- Choosing appropriate indexes

These optimizations make your workflows faster, especially when scaling up to large datasets or preparing for production deployments.

12.2 Vectorized Operations vs Loops

When working with pandas, one of the most effective ways to write fast, readable, and scalable code is by using **vectorized operations** instead of traditional Python loops. Vectorization refers to applying operations over entire arrays (or Series/DataFrames) at once using optimized, low-level implementations—typically backed by NumPy.

12.2.1 Why Avoid Loops in Pandas?

Loops in Python (like `for` or `while`) are slow because they process one element at a time and operate in the interpreted layer. In contrast, pandas and NumPy vectorized methods run compiled C code under the hood, making them significantly faster.

Let's look at an example that illustrates the difference.

12.2.2 Example: Adding Two Columns

With a Loop (Slow):

Full runnable code:

```
import pandas as pd
import numpy as np
import time

# Sample DataFrame
df = pd.DataFrame({
    'a': np.random.randint(0, 100, size=100000),
    'b': np.random.randint(0, 100, size=100000)
})

# Using loop
start = time.time()
df['sum_loop'] = [x + y for x, y in zip(df['a'], df['b'])]
end = time.time()
print("Loop time:", round(end - start, 4), "seconds")
```

Vectorized (Fast):

```
start = time.time()
df['sum_vectorized'] = df['a'] + df['b']
end = time.time()
print("Vectorized time:", round(end - start, 4), "seconds")
```

In practice, the vectorized operation can be **10–100 times faster** than the loop, especially as dataset size increases.

12.2.3 More Examples: Vectorizing Common Tasks

Squaring Values

Loop:

```
df['squared_loop'] = [x**2 for x in df['a']]
```

Vectorized:

```
df['squared_vec'] = df['a'] ** 2
```

Conditional Column

Suppose you want to label values as "high" if > 50, otherwise "low":

With Loop:

```
df['label_loop'] = ['high' if x > 50 else 'low' for x in df['a']]
```

Vectorized with `np.where()`:

```
df['label_vec'] = np.where(df['a'] > 50, 'high', 'low')
```

Row-wise Mean (Avoid `for`)

Don't loop row by row—use vectorized methods:

```
df['mean_val'] = df[['a', 'b']].mean(axis=1)
```

12.2.4 Summary Table

Task	Loop-Based Code	Vectorized Code
Add columns	`[x + y for x, y in zip(...)]`	`df['a'] + df['b']`
Square values	`[x**2 for x in df['a']]`	`df['a'] ** 2`
Conditional label	`['high' if x > 50 else 'low' ...]`	`np.where(df['a'] > 50, ...)`
Row-wise stats	loop each row manually	`df.mean(axis=1)`

12.2.5 Final Thoughts

Vectorized operations not only provide **significant performance improvements**, especially on large datasets, but also make your code **cleaner and more concise**. Always prefer built-in pandas and NumPy methods for transformations and avoid Python loops unless absolutely necessary. When profiling your code, if it's slow, loops are a good place to start optimizing.

12.3 Using `eval()` and `query()` for Speed

When working with large DataFrames, performance can become a bottleneck, especially during filtering or when computing new columns with complex expressions. Pandas offers two powerful tools—`eval()` and `query()`—to write cleaner, faster, and more memory-efficient code by leveraging optimized expression evaluation using `numexpr` under the hood.

12.3.1 What is `eval()`?

`DataFrame.eval()` evaluates string expressions using column names directly. It's particularly useful for computing new columns or performing arithmetic operations.

Basic Usage:

Full runnable code:

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'a': np.random.randint(1, 100, size=1000000),
    'b': np.random.randint(1, 100, size=1000000),
    'c': np.random.rand(1000000)
})

# Traditional method
df['result'] = df['a'] + df['b'] * df['c']

# Using eval (faster and more concise)
df.eval('result_eval = a + b * c', inplace=True)
```

This avoids the overhead of parsing Python objects repeatedly and improves performance when working with large datasets.

12.3.2 What is `query()`?

`DataFrame.query()` allows you to filter rows based on string conditions using a SQL-like syntax, making your code both faster and more readable.

Example: Filtering Rows

```python
# Traditional filtering
filtered = df[(df['a'] > 50) & (df['b'] < 30)]

# Using query (faster and cleaner)
filtered_q = df.query('a > 50 and b < 30')
```

This approach reduces both verbosity and execution time, especially for large data.

12.3.3 Why Use `eval()` and `query()`?

- **Performance**: Internally uses `numexpr` for fast computation on large arrays.
- **Memory-efficient**: Reduces intermediate object creation.
- **Readability**: Cleaner syntax, especially for complex logic.

- **Convenience**: No need to repeat `df['col']`—just use the column names directly.

12.3.4 Practical Examples

Creating New Columns

```
# Compute a weighted score
df.eval('score = 0.6 * a + 0.4 * b', inplace=True)
```

Filtering with Conditions

```
# Find rows where score is above 70 and 'c' is below 0.5
high_scores = df.query('score > 70 and c < 0.5')
```

Comparing Performance

Full runnable code:

```
import time

# Loop-style filtering
start = time.time()
df[(df['a'] > 50) & (df['b'] < 20)]
print("Loop-style:", round(time.time() - start, 4), "s")

# Query-style filtering
start = time.time()
df.query('a > 50 and b < 20')
print("Query-style:", round(time.time() - start, 4), "s")
```

12.3.5 Limitations and Tips

- Both `eval()` and `query()` work best with simple arithmetic and logical expressions.
- Use `@` to reference Python variables inside queries:

  ```
  threshold = 50
  df.query('a > @threshold')
  ```

- They may not support complex Python functions or method chains (e.g., `.apply()`).

12.3.6 Conclusion

`eval()` and `query()` are excellent tools for optimizing pandas workflows involving computation and filtering. They help reduce code verbosity and improve performance, especially on large datasets. When used judiciously, they can simplify your data transformation pipeline without sacrificing speed.

12.4 Practical Examples: Speeding Up Data Processing

In this section, we'll walk through a realistic scenario that highlights how performance optimization techniques—like memory reduction, vectorized operations, and use of `eval()` and `query()`—can drastically speed up data processing in pandas. We'll also introduce basic profiling tools to assess performance improvements.

12.4.1 Scenario: Processing Retail Transactions

Imagine a dataset containing millions of rows representing retail transactions. We'll optimize the workflow step-by-step.

Step 1: Simulate a Large Dataset

Full runnable code:

```
import pandas as pd
import numpy as np

# Simulate 1 million transactions
np.random.seed(0)
n = 1_000_000
df = pd.DataFrame({
    'customer_id': np.random.randint(1000, 9999, size=n),
    'product_id': np.random.randint(1, 100, size=n),
    'quantity': np.random.randint(1, 5, size=n),
    'unit_price': np.random.uniform(10, 100, size=n),
    'category': np.random.choice(['A', 'B', 'C', 'D'], size=n),
    'date': pd.date_range('2023-01-01', periods=n, freq='min')
})

print(df.info(memory_usage='deep'))
```

This shows high memory usage due to large integer and object columns.

12.4.2 Step 2: Reduce Memory Usage

```
df['customer_id'] = df['customer_id'].astype('int32')
df['product_id'] = df['product_id'].astype('int16')
df['quantity'] = df['quantity'].astype('int8')
df['category'] = df['category'].astype('category')

print(df.info(memory_usage='deep'))
```

Memory savings: Reducing column types from `int64/object` to smaller numerics or `category` can cut memory by over 50%.

12.4.3 Step 3: Use Vectorized Operations

Instead of using loops to compute total transaction value, do:

```
df['total_value'] = df['quantity'] * df['unit_price']
```

This vectorized operation is much faster than iterating row by row with `.apply()` or a `for` loop.

12.4.4 Step 4: Use `eval()` and `query()` for Speed

```
# Faster expression evaluation
df.eval('total_eval = quantity * unit_price', inplace=True)

# Efficient filtering with query
high_value = df.query('total_value > 200 and category == "A"')
```

This approach avoids temporary objects and speeds up calculations.

12.4.5 Step 5: Profiling and Benchmarking

You can use `%timeit` in Jupyter or `time` module to compare performance:

```
import time

# Traditional filter
start = time.time()
df[(df['total_value'] > 200) & (df['category'] == 'A')]
print("Loop-style filter:", round(time.time() - start, 4), "s")

# Query-style filter
start = time.time()
df.query('total_value > 200 and category == "A"')
```

```
print("Query-style filter:", round(time.time() - start, 4), "s")
```

Expect the `query()` method to consistently outperform the traditional syntax for large datasets.

12.4.6 Summary and Best Practices

- **Convert dtypes** early to reduce memory.
- **Prefer vectorized operations** over loops or `.apply()`.
- **Use `eval()` and `query()`** for large, repeated expressions or filters.
- **Profile your code** with `%timeit`, `memory_usage()`, and `.info()` to spot inefficiencies.

> WARNING **Caution**: For small datasets, these optimizations may offer negligible gains. Use them where scale and performance matter most.

By applying these techniques thoughtfully, you can process millions of rows efficiently and make your data pipelines more robust and scalable.

Chapter 13.

Advanced Indexing and MultiIndex

13 Advanced Indexing and MultiIndex

13.1 Creating and Using MultiIndex DataFrames

Pandas' **MultiIndex** (also known as hierarchical indexing) is a powerful feature that allows you to represent higher-dimensional data within a two-dimensional `DataFrame`. Rather than relying on a single index column, MultiIndex allows for **multiple levels of indexing**, both on rows and columns. This structure is especially helpful in fields like finance, sensor analytics, and survey data—where data is often naturally hierarchical.

13.1.1 Why Use MultiIndex?

MultiIndex provides a way to:

- Efficiently store and query **grouped or nested data**
- Represent data in **panel or 3D-like form** without increasing dimensionality
- Simplify **grouped operations**, aggregations, and reshaping

However, this power comes with some complexity: MultiIndexed DataFrames can be harder to manipulate and visualize at first glance.

13.1.2 Creating a MultiIndex

There are several ways to create a MultiIndex:

From Tuples

You can define hierarchical keys using a list of tuples:

Full runnable code:

```
import pandas as pd

index = [('North', 2023), ('North', 2024),
         ('South', 2023), ('South', 2024)]
multi_idx = pd.MultiIndex.from_tuples(index, names=['Region', 'Year'])

data = [100, 120, 90, 110]
df = pd.DataFrame(data, index=multi_idx, columns=['Sales'])

print(df)
```

Output:

```
              Sales
```

```
Region Year
North  2023    100
       2024    120
South  2023     90
       2024    110
```

Here, `Region` and `Year` are index levels. This structure lets you access or slice data in more flexible ways.

From Arrays

You can also create a MultiIndex from separate lists or arrays:

```python
regions = ['East', 'East', 'West', 'West']
years = [2023, 2024, 2023, 2024]

multi_idx = pd.MultiIndex.from_arrays([regions, years], names=['Region', 'Year'])
df = pd.DataFrame({'Profit': [30, 50, 40, 60]}, index=multi_idx)

print(df)
```

From Existing DataFrame with `set_index()`

Another common method is to convert columns into a MultiIndex:

```python
data = {
    'Region': ['North', 'North', 'South', 'South'],
    'Year': [2023, 2024, 2023, 2024],
    'Sales': [200, 220, 180, 190]
}

df = pd.DataFrame(data)
df = df.set_index(['Region', 'Year'])

print(df)
```

This method is useful when importing structured data from external sources and transforming it for analysis.

13.1.3 Inspecting and Navigating MultiIndex

You can explore and manipulate the index with:

```python
df.index.levels          # Show unique values in each level
df.index.names           # Index level names
df.loc['North']          # All years for 'North'
df.loc[('South', 2023)] # Specific selection
```

You can also reset the index:

```python
df_reset = df.reset_index()
```

This will flatten the hierarchy back into columns.

13.1.4 Benefits of MultiIndex

- **Compact storage** for grouped data
- Supports complex **groupby** operations
- Enables **reshaping** with `.unstack()` and `.stack()`

13.1.5 Trade-Offs

- More complex to slice and filter than flat indices
- Certain operations may behave differently or require more syntax
- Not always optimal for small/simple datasets

13.1.6 Summary

The MultiIndex structure in pandas is a foundational tool for managing hierarchical or grouped data within a 2D frame. Whether you're organizing sales across regions and years, comparing experiments across test conditions, or analyzing panel data, MultiIndex allows you to preserve the dimensionality of your data while enabling powerful operations and aggregations.

In the next section, you'll learn how to **slice and select** data within a MultiIndex using intuitive and advanced methods.

13.2 Index Slicing and Advanced Selections

Once you've created a `MultiIndex` DataFrame, the next step is learning how to access and manipulate its data efficiently. This section explores several powerful techniques: accessing data using `.loc[]` with tuples and slices, using the `.xs()` (cross-section) method, and performing partial and boolean selections.

13.2.1 Accessing with `.loc[]` and Tuples

In MultiIndex DataFrames, you use **tuples** in `.loc[]` to refer to specific index combinations:

Full runnable code:

```python
import pandas as pd

index = pd.MultiIndex.from_tuples(
    [('North', 2023), ('North', 2024),
     ('South', 2023), ('South', 2024)],
    names=['Region', 'Year']
)

df = pd.DataFrame({'Sales': [100, 120, 90, 110]}, index=index)
print(df.loc[('North', 2023)])
```

Output:

```
Sales    100
Name: (North, 2023), dtype: int64
```

To select all rows for a specific outer level (e.g., all years in the “North” region):

```python
print(df.loc['North'])
```

13.2.2 Using Slices in `.loc[]`

You can also use Python’s `slice()` to select a range of index values:

```python
print(df.loc[('North', slice(2023, 2024))])
```

Or to access all data across regions for a particular year:

```python
print(df.loc[(slice(None), 2023)])
```

To ensure readable syntax, consider using `pd.IndexSlice`:

```python
idx = pd.IndexSlice
print(df.loc[idx[:, 2023], :])
```

This selects all regions for the year 2023.

13.2.3 The `.xs()` Method: Cross-Section Selections

The `.xs()` method is ideal for slicing across index levels cleanly, especially when working with multiple levels. For example:

```python
# Get all data for the year 2024
print(df.xs(2024, level='Year'))
```

You can also select along columns (if you have a column MultiIndex) by setting `axis=1`.

13.2.4 Partial Indexing (Shorthand)

Pandas allows **partial indexing** when the index is sorted. If your index is sorted properly, you can omit levels:

```
df_sorted = df.sort_index()
print(df_sorted.loc['South'])
```

This returns all years for `"South"`. Keep in mind this only works if the index is **lexsorted**—you can check this with:

```
df.index.is_lexsorted()  # Deprecated; use df.index.is_monotonic_increasing in newer versions
```

Or sort with:

```
df = df.sort_index()
```

13.2.5 Boolean Selection with MultiIndex

You can still use boolean filters on MultiIndex DataFrames. Here's how to select rows with `Sales > 100`:

```
print(df[df['Sales'] > 100])
```

Or filter using index levels:

```
print(df[df.index.get_level_values('Region') == 'North'])
```

You can combine these filters too:

```
mask = (df.index.get_level_values('Region') == 'North') & (df['Sales'] > 110)
print(df[mask])
```

13.2.6 Summary

MultiIndex slicing and selection gives you immense control over complex, hierarchical data. Use `.loc[]` with tuples and slices for direct access, `.xs()` for concise cross-sectional views, and boolean logic for conditional filtering. With these tools, you can efficiently query large and structured datasets in a flexible and readable manner.

In the next section, we'll learn how to reshape MultiIndexed DataFrames using `stack()` and `unstack()`—essential for converting between wide and long formats.

13.3 Stacking and Unstacking DataFrames

One of the key strengths of Pandas is its ability to reshape datasets using hierarchical (MultiIndex) structures. Two powerful tools for this are the `stack()` and `unstack()` methods. These allow you to pivot your data between wide and long formats, enabling flexible data transformation for analysis or visualization.

13.3.1 What Are stack() and unstack()?

- **`stack()`**: Moves the **columns** into the **row index** (i.e., pivots columns down into the index).
- **`unstack()`**: Moves a **level of the index** into the **columns** (i.e., pivots row index into columns).

Both work best on DataFrames with MultiIndexes, either in the index, columns, or both.

13.3.2 Example: Basic Stack and Unstack

Start with a simple MultiIndex DataFrame:

Full runnable code:

```python
import pandas as pd

data = {
    'Q1': [200, 210, 190, 220],
    'Q2': [250, 260, 230, 240]
}

index = pd.MultiIndex.from_product(
    [['North', 'South'], ['2023', '2024']],
    names=['Region', 'Year']
)

df = pd.DataFrame(data, index=index)
print(df)
```

Output:

```
              Q1   Q2
Region Year
North  2023  200  250
       2024  210  260
South  2023  190  230
       2024  220  240
```

13.3.3 Stacking: Wide Long

Now let's pivot the columns into a new inner index level:

```
stacked = df.stack()
print(stacked)
```

Output:

```
Region  Year  Quarter
North   2023  Q1          200
               Q2          250
        2024  Q1          210
               Q2          260
South   2023  Q1          190
               Q2          230
        2024  Q1          220
               Q2          240
dtype: int64
```

The result is a **Series** with a 3-level MultiIndex: `Region`, `Year`, and `Quarter`.

If you need to keep it as a DataFrame:

```
stacked_df = df.stack().to_frame(name='Sales')
print(stacked_df)
```

13.3.4 Unstacking: Long Wide

You can reverse the operation by unstacking:

```
unstacked = stacked.unstack()
print(unstacked)
```

You can also specify which level to unstack (e.g., 'Region'):

```
print(stacked.unstack(level='Region'))
```

Or unstack multiple levels:

```
print(stacked.unstack(['Region', 'Quarter']))
```

13.3.5 When to Use These

- Use `stack()` to **compress** wide tables for statistical modeling or export to long-format tools (e.g., Seaborn).
- Use `unstack()` to **spread out** data, making it easier to visualize comparisons or pivot

for reporting.

13.3.6 Practical Reshaping Use Case

Imagine you're analyzing quarterly sales and want to compare `Q1` vs `Q2` across regions and years side-by-side:

```python
# From long format
df_long = df.stack().reset_index()
df_long.columns = ['Region', 'Year', 'Quarter', 'Sales']

# Pivot to wide format
reshaped = df_long.pivot_table(
    index=['Region', 'Year'],
    columns='Quarter',
    values='Sales'
)

print(reshaped)
```

13.3.7 Summary

The `stack()` and `unstack()` methods are essential for reshaping DataFrames, especially those with hierarchical structures. They help pivot between long and wide formats depending on your analytical or visualization needs. Mastering these methods will let you adapt data structures fluidly, unlocking deeper insights from complex datasets.

Next, we'll apply these transformations to real-world hierarchical data in a full analysis example.

13.4 Practical Examples: Hierarchical Data Analysis

MultiIndex DataFrames are invaluable for managing complex datasets involving multiple grouping variables, time series, or panel data. By leveraging hierarchical indexes, you can efficiently organize, analyze, and visualize data that spans several dimensions. In this section, we'll explore real-world scenarios where MultiIndexes simplify data handling and insight generation.

13.4.1 Example 1: Financial Time Series with Multiple Assets

Imagine you have daily stock price data for multiple companies over several months. Instead of keeping separate DataFrames, a MultiIndex lets you combine and analyze them easily.

Full runnable code:

```python
import pandas as pd
import numpy as np

# Simulate stock prices for 3 companies over 5 days
dates = pd.date_range('2024-01-01', periods=5)
companies = ['Apple', 'Google', 'Amazon']

# Create MultiIndex from product of companies and dates
index = pd.MultiIndex.from_product([companies, dates], names=['Company', 'Date'])

# Random stock prices
np.random.seed(0)
prices = np.random.uniform(100, 500, size=len(index))

# Build DataFrame
stock_df = pd.DataFrame({'Price': prices}, index=index)
print(stock_df.head(10))
```

Output:

```
                  Price
Company Date
Apple   2024-01-01  344.14
        2024-01-02  404.29
        2024-01-03  456.12
        2024-01-04  348.55
        2024-01-05  259.46
Google  2024-01-01  455.07
        2024-01-02  319.18
        2024-01-03  305.91
        2024-01-04  485.20
        2024-01-05  236.92
```

Grouping and Aggregation

You can group by one level (e.g., company) to compute average prices:

```python
avg_prices = stock_df.groupby(level='Company').mean()
print(avg_prices)
```

Output:

```
               Price
Company
Amazon      298.02
Apple       362.91
```

```
Google      360.06
```

Or analyze price changes over time for each company:

```
daily_returns = stock_df.groupby(level='Company').Price.pct_change()
stock_df['Daily Return'] = daily_returns
print(stock_df.head(10))
```

13.4.2 Example 2: Multi-Category Sales Data

Suppose you have sales data categorized by region, product category, and month:

```
regions = ['North', 'South']
categories = ['Electronics', 'Furniture']
months = ['2023-01', '2023-02']

index = pd.MultiIndex.from_product([regions, categories, months],
                                   names=['Region', 'Category', 'Month'])
sales = np.random.randint(1000, 5000, size=len(index))

sales_df = pd.DataFrame({'Sales': sales}, index=index)
print(sales_df)
```

Aggregating Across Levels

Calculate total sales by region (ignoring category and month):

```
total_by_region = sales_df.groupby(level='Region').sum()
print(total_by_region)
```

Or sales by category across all regions and months:

```
total_by_category = sales_df.groupby(level='Category').sum()
print(total_by_category)
```

13.4.3 Visualization with MultiIndex DataFrames

Pandas plotting works seamlessly with MultiIndexes. For example, plot monthly sales for each category in the North region:

```
import matplotlib.pyplot as plt

north_sales = sales_df.loc['North'].unstack(level='Month')
north_sales.plot(kind='bar')
plt.title('Monthly Sales by Category in North Region')
plt.xlabel('Category')
plt.ylabel('Sales')
plt.show()
```

This visualization clearly contrasts sales trends for Electronics and Furniture over months.

13.4.4 Why Use MultiIndex?

- **Organizes complex data** intuitively along multiple dimensions.
- Enables **fast group-based computations** and slicing.
- Facilitates reshaping for **reporting and visualization**.
- Maintains a **compact, tabular format** for panel data and time series.

13.4.5 Tips for Working with MultiIndex

- Use `.loc` with tuples for precise slicing: `df.loc[('North', 'Electronics')]`.
- Reset index if flat DataFrame needed for export or simpler operations.
- Combine with `stack()` and `unstack()` to pivot data as needed.

13.4.6 Summary

MultiIndex DataFrames empower you to analyze hierarchical datasets like financial time series and multi-category sales data efficiently. By grouping, aggregating, and visualizing along different index levels, you unlock richer insights without cumbersome manual reshaping. Practice these techniques on your data to harness the full power of Pandas' hierarchical indexing.

Chapter 14.

Customizing Pandas with Extensions and APIs

14 Customizing Pandas with Extensions and APIs

14.1 Creating Custom Accessors and Extensions

Pandas is a powerful library, but sometimes your project demands specialized functionality tailored to your domain or workflow. To accommodate this, pandas offers **extension points** — ways to add your own custom methods to DataFrames or Series through **custom accessors**. These allow you to enrich pandas objects with domain-specific features while keeping the core API clean and consistent.

14.1.1 What Are Custom Accessors?

A **custom accessor** is a user-defined attribute attached to a pandas `DataFrame` or `Series` that groups related methods together. For example, pandas provides `.str` for string operations or `.dt` for datetime methods. You can create your own accessor, such as `.geo` for geographic computations or `.finance` for financial calculations, enabling intuitive syntax like:

```
df.geo.calculate_distance()
df.finance.calculate_roi()
```

This makes your code cleaner and your extensions discoverable through auto-completion.

14.1.2 How to Create a Custom DataFrame Accessor

Pandas provides a decorator to register your accessor:

```
@pd.api.extensions.register_dataframe_accessor("your_accessor_name")
```

Your accessor class must:

- Take the pandas object (DataFrame or Series) as input on initialization.
- Store the object (usually as `self._obj`).
- Define any methods you want to expose under the accessor namespace.

14.1.3 Step-by-Step Example: A Simple Accessor for DataFrame Summary

Let's create a custom accessor called `.summary` that provides an overview of numeric columns, including their mean, median, and standard deviation.

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("summary")
```

```python
class SummaryAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    def stats(self):
        """Return a DataFrame with mean, median, and std for numeric columns."""
        numeric = self._obj.select_dtypes(include='number')
        summary_df = pd.DataFrame({
            'mean': numeric.mean(),
            'median': numeric.median(),
            'std': numeric.std()
        })
        return summary_df
```

14.1.4 Using the Custom Accessor

Create a sample DataFrame and try the new `.summary.stats()` method:

```python
data = {
    'A': [10, 20, 30, 40],
    'B': [1.5, 2.5, 3.5, 4.5],
    'C': ['x', 'y', 'z', 'w']  # Non-numeric column
}
df = pd.DataFrame(data)

print(df.summary.stats())
```

Output:

```
        mean  median        std
A  25.000000    25.0  12.909944
B   3.000000     3.0   1.290994
```

Notice that only numeric columns are included in the summary, and the accessor neatly groups related functionality.

14.1.5 Benefits of Custom Accessors

- **Namespace clarity**: Keep your extensions grouped under meaningful attributes rather than cluttering the global namespace.
- **Reusability**: Package custom accessors as modules and share across projects.
- **Integration**: Work seamlessly with pandas objects, preserving the natural API feel.
- **Discoverability**: Users can find your custom methods via standard attribute access and tab completion.

14.1.6 Tips and Considerations

- You can also create **Series accessors** with `@pd.api.extensions.register_series_accessor`.
- Accessors should **not mutate** the original DataFrame directly. Instead, return new DataFrames or values.
- Ensure your methods handle edge cases (empty DataFrames, missing data).
- You can bundle multiple related methods in one accessor for richer domain-specific APIs.

14.1.7 Summary

Custom accessors unlock powerful ways to extend pandas with your own tailored methods, enhancing readability and maintainability. By using `@pd.api.extensions.register_dataframe_accessor`, you can easily inject domain-specific logic directly into the pandas API, making your data analysis workflows more expressive and efficient.

14.2 Using Pandas with NumPy and Other Libraries

Pandas is built on top of NumPy, which means it naturally integrates with NumPy arrays and many other popular Python data science libraries like SciPy, statsmodels, and scikit-learn. Understanding how to leverage this interoperability lets you combine pandas' rich data manipulation capabilities with powerful numerical, statistical, and machine learning tools efficiently.

14.2.1 Pandas and NumPy: Seamless Interchange

At its core, a pandas `Series` or `DataFrame` stores data in NumPy arrays. You can access the underlying NumPy array with the `.values` or `.to_numpy()` methods:

Full runnable code:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

# Extract NumPy array
arr = df.values
```

```
print(arr)
# Output:
# [[1 4]
#  [2 5]
#  [3 6]]

# Using .to_numpy() (recommended for newer code)
arr2 = df.to_numpy()
print(arr2)
```

This seamless exchange allows you to use pandas data directly with NumPy functions for fast numerical computations:

```
# Compute row sums with NumPy
row_sums = np.sum(df.to_numpy(), axis=1)
print(row_sums)  # Output: [5 7 9]
```

14.2.2 When to Use Pandas vs NumPy

- Use **pandas** for **labeled** data manipulation, easy filtering, missing data handling, and integration with heterogeneous data types.
- Use **NumPy** when you need fast, vectorized numeric computations on homogeneous data arrays, or for functions not yet available in pandas.

Example: Computing rolling means is straightforward in pandas with `.rolling()`, but complex linear algebra routines require NumPy or SciPy.

14.2.3 Interoperability with SciPy and statsmodels

SciPy and statsmodels focus on advanced statistical and scientific computing. These libraries often expect NumPy arrays as input, but pandas data can be converted easily:

```
from scipy import stats
import statsmodels.api as sm

# Using pandas Series with SciPy
data = df['A']
z_scores = stats.zscore(data)
print(z_scores)

# Using DataFrame with statsmodels for regression
X = sm.add_constant(df['A'])  # Add intercept term
model = sm.OLS(df['B'], X).fit()
print(model.summary())
```

Passing pandas Series or DataFrames directly to these libraries typically works well, but be mindful that some functions might drop labels and return NumPy arrays, requiring you to handle indices manually afterward.

14.2.4 Using Pandas with scikit-learn

scikit-learn is the go-to library for machine learning in Python. While it expects NumPy arrays as input, pandas DataFrames can be passed directly without issue, retaining column names for convenience in later steps (e.g., feature importance):

```
from sklearn.linear_model import LinearRegression

X = df[['A']]  # Predictor
y = df['B']    # Target

model = LinearRegression()
model.fit(X, y)

print("Coefficient:", model.coef_)
print("Intercept:", model.intercept_)
```

Here, pandas DataFrames and Series act as natural containers for features and targets, but under the hood, scikit-learn works with NumPy arrays.

14.2.5 Converting Between pandas and Other Libraries

- **Pandas → NumPy**: Use `.to_numpy()` for efficient conversion.
- **NumPy → pandas**: Use `pd.Series()` or `pd.DataFrame()` with optional index and columns.

Example:

```
arr = np.array([10, 20, 30])
series = pd.Series(arr, index=['a', 'b', 'c'])
print(series)
```

14.2.6 Summary

- Pandas is tightly integrated with NumPy, enabling fast numerical operations on labeled data.
- Use pandas for convenient data wrangling; NumPy, SciPy, statsmodels, and scikit-learn for specialized scientific, statistical, or machine learning tasks.
- Convert pandas objects to NumPy arrays using `.to_numpy()` when needed.
- Passing pandas data directly to external libraries is usually seamless, but watch out for losing index/column labels.

Leveraging these libraries together lets you build powerful and efficient data science workflows combining the best features of each tool.

14.3 Pandas API for Window and Rolling Functions

Pandas offers powerful window functions that allow you to compute statistics over sliding or expanding windows in your data. These functions are essential for time series analysis, smoothing noisy data, detecting anomalies, and calculating financial indicators. The key window methods are `.rolling()`, `.expanding()`, and `.ewm()`, each suited to different use cases.

14.3.1 `.rolling()`: Moving Window Calculations

The `.rolling()` method creates a rolling window over your data, allowing you to apply aggregation functions like mean, sum, or custom functions on fixed-size subsets.

Key parameters:

- `window`: size of the moving window (number of observations)
- `min_periods`: minimum number of observations required in window to have a value
- `center`: whether the label is at the center or right edge of the window (default is right)

Example: Moving average smoothing

```python
import pandas as pd
import numpy as np

dates = pd.date_range('2024-01-01', periods=10)
data = pd.Series([2, 3, 5, 8, 13, 21, 34, 55, 89, 144], index=dates)

# 3-day moving average
rolling_mean = data.rolling(window=3).mean()
print(rolling_mean)
```

This smooths the series by averaging every 3 consecutive values, which is useful to reduce noise or highlight trends.

14.3.2 `.expanding()`: Cumulative Calculations

The `.expanding()` method computes aggregates over an expanding window, starting from the first element and including all prior data points up to the current one.

Example: Cumulative sum and mean

```python
cumulative_sum = data.expanding(min_periods=1).sum()
cumulative_mean = data.expanding(min_periods=1).mean()

print(cumulative_sum)
print(cumulative_mean)
```

This is useful for understanding cumulative totals or averages over time, for example, cumulative sales or average temperature.

14.3.3 `.ewm()`: Exponentially Weighted Functions

The `.ewm()` method calculates exponentially weighted metrics, which give more weight to recent observations and less to older ones. This is particularly useful for financial indicators and anomaly detection where recent values are more important.

Key parameters:

- `span`, `halflife`, or `alpha`: control the decay of weights
- `adjust`: whether to adjust weights (default True)

Example: Exponential weighted moving average (EWMA)

```
ewm_mean = data.ewm(span=3, adjust=False).mean()
print(ewm_mean)
```

EWMA reacts faster to recent changes compared to a simple moving average, which makes it useful for tracking trends and volatility in stock prices.

14.3.4 Practical Use Cases

- **Smoothing noisy sensor data:** Use `.rolling()` mean or median to smooth out fluctuations.
- **Financial analysis:** Calculate moving averages, EWMA, and rolling volatility to identify trends and risks.
- **Anomaly detection:** Identify points that deviate significantly from rolling averages or exponentially weighted means.
- **Cumulative metrics:** Track total sales or accumulated values over time using `.expanding()`.

14.3.5 Combining Window Functions with Other Operations

Window functions return pandas objects, so you can chain methods:

```
# 7-day rolling average, then difference
data.rolling(window=7).mean().diff()
```

This computes a smoothed series and then its change, useful for momentum analysis.

14.3.6 Summary

- `.rolling(window)`: Fixed-size moving window for calculations like mean, sum, std.
- `.expanding()`: Growing window starting at the first data point for cumulative stats.
- `.ewm()`: Exponential weighting to prioritize recent data.
- Use these window functions to smooth data, detect anomalies, calculate financial indicators, and summarize evolving statistics.

Mastering these tools will empower you to extract meaningful insights from time-dependent data efficiently and effectively.

14.4 Practical Examples: Extending Pandas Functionality

In this section, we'll explore a practical project where creating custom pandas extensions and leveraging advanced APIs enhances your data workflow. By building reusable utilities and domain-specific validations, you can streamline complex data tasks, reduce errors, and improve code readability.

14.4.1 Scenario: Financial Data Validation and Transformation

Suppose you regularly work with financial datasets containing columns like `"Date"`, `"Price"`, and `"Volume"`. You want to:

- Validate data integrity (e.g., no negative prices or volumes).
- Add domain-specific transformations (e.g., calculate daily returns).
- Easily reuse these functions across multiple DataFrames.

Custom pandas accessors and extension APIs allow you to embed these capabilities directly on your DataFrames, making your workflow elegant and consistent.

14.4.2 Step 1: Create a Custom Accessor for Financial Data

Use the `@pd.api.extensions.register_dataframe_accessor` decorator to add a `.finance` namespace:

```python
import pandas as pd
import numpy as np

@pd.api.extensions.register_dataframe_accessor("finance")
class FinanceAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj
    def validate(self):
        """Check for negative prices or volumes."""
        if (self._obj['Price'] < 0).any():
            raise ValueError("Negative prices found!")
        if (self._obj['Volume'] < 0).any():
            raise ValueError("Negative volumes found!")
        print("Validation passed: No negative values.")

    def daily_returns(self):
        """Calculate daily returns from Price column."""
        returns = self._obj['Price'].pct_change()
        return returns.fillna(0)

    def add_daily_returns(self):
        """Add daily returns as a new column."""
        self._obj['Daily_Return'] = self.daily_returns()
        return self._obj
```

14.4.3 Step 2: Use the Custom Accessor in Practice

Create a sample financial DataFrame:

```
data = {
    'Date': pd.date_range('2024-01-01', periods=5),
    'Price': [100, 102, 101, 105, 107],
    'Volume': [200, 220, 210, 230, 240]
}
df = pd.DataFrame(data).set_index('Date')
```

Now use the `.finance` accessor to validate and add daily returns:

```
df.finance.validate()
df = df.finance.add_daily_returns()
print(df)
```

Output:

```
Validation passed: No negative values.
            Price  Volume  Daily_Return
Date
2024-01-01    100     200      0.000000
2024-01-02    102     220      0.020000
2024-01-03    101     210     -0.009804
2024-01-04    105     230      0.039604
2024-01-05    107     240      0.019048
```

14.4.4 Step 3: Integrate with External Tools (NumPy & Matplotlib)

You can also extend functionality by integrating NumPy for custom statistics or Matplotlib for quick visualizations within your accessor:

```
import matplotlib.pyplot as plt

@pd.api.extensions.register_dataframe_accessor("finance")
class FinanceAccessor:
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    # (previous methods omitted for brevity)

    def volatility(self, window=3):
        """Calculate rolling volatility (std dev of returns)."""
        returns = self.daily_returns()
        vol = returns.rolling(window=window).std()
        return vol.fillna(0)

    def plot_price_and_returns(self):
        """Plot Price and Daily Returns on twin axes."""
        fig, ax1 = plt.subplots(figsize=(8,4))
        ax1.set_title("Price and Daily Returns")
        ax1.plot(self._obj.index, self._obj['Price'], 'b-', label='Price')
        ax1.set_ylabel('Price', color='b')

        ax2 = ax1.twinx()
        ax2.plot(self._obj.index, self._obj['Daily_Return'], 'r--', label='Daily Return')
        ax2.set_ylabel('Return', color='r')

        plt.show()
```

Usage:

```
df.finance.add_daily_returns()
volatility = df.finance.volatility()
print("Volatility:\n", volatility)

df.finance.plot_price_and_returns()
```

14.4.5 Benefits of This Approach

- **Encapsulation**: Group domain-specific methods logically, avoiding scattered functions.
- **Reusability**: Easily reuse and share custom accessors across projects.
- **Readability**: Improve code clarity by calling `.finance.method()` on any DataFrame.
- **Integration**: Seamlessly combine pandas data manipulation with external libraries like NumPy and Matplotlib.

14.4.6 Final Thoughts

Extending pandas with custom accessors transforms repetitive and complex operations into simple, expressive commands tailored to your domain. Combine this with pandas' built-in APIs and external libraries to build robust, maintainable data pipelines.

Chapter 15.

Real-World Data Science Projects with Pandas

1. Project 1: Exploratory Data Analysis on a Public Dataset
2. Project 2: Time Series Forecasting Preparation
3. Project 3: Customer Segmentation with Grouping and Aggregation
4. Project 4: Data Cleaning and Merging from Multiple Sources
5. Project 5: End-to-End Data Science Pipeline using Pandas

15 Real-World Data Science Projects with Pandas

15.1 Project 1: Exploratory Data Analysis on a Public Dataset

Exploratory Data Analysis (EDA) is a crucial first step in any data science project. It helps you understand the dataset's structure, detect anomalies, and generate hypotheses. In this project, we'll walk through EDA on the classic Titanic dataset using pandas. The dataset contains passenger details such as age, sex, ticket class, and survival status — perfect for demonstrating pandas' power in data exploration.

15.1.1 Step 1: Loading the Dataset

First, import pandas and load the Titanic dataset. You can get it directly from the seaborn library for convenience:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load Titanic dataset from seaborn
df = sns.load_dataset('titanic')
print(df.head())
```

This dataset has columns like `survived`, `pclass`, `sex`, `age`, and more. Let's start by understanding the shape and basic info.

```
print(f"Dataset shape: {df.shape}")
print(df.info())
```

15.1.2 Step 2: Data Cleaning and Handling Missing Values

Check for missing values using `.isna()` and `.sum()`:

```
print(df.isna().sum())
```

You'll notice significant missing data in `age`, `deck`, and `embarked`. For simplicity:

- Fill missing `age` values with the median age.
- Drop the `deck` column due to too many missing values.
- Fill missing `embarked` values with the mode.

```
df['age'].fillna(df['age'].median(), inplace=True)
df.drop(columns=['deck'], inplace=True)
df['embarked'].fillna(df['embarked'].mode()[0], inplace=True)
```

Verify no missing data remains in key columns:

```
print(df[['age', 'deck', 'embarked']].isna().sum())
```

15.1.3 Step 3: Summarizing the Data

Use `.describe()` to get statistical summaries of numeric columns:

```
print(df.describe())
```

This reveals the distribution of age, fare, and more. For categorical data, use `.value_counts()`:

```
print(df['sex'].value_counts())
print(df['pclass'].value_counts())
print(df['embarked'].value_counts())
```

Understanding the distribution of classes and embarkation points provides insights into passenger demographics.

15.1.4 Step 4: Grouping and Aggregation

Now, let's analyze survival rates by different groups:

```
# Survival rate overall
print(f"Overall survival rate: {df['survived'].mean():.2f}")

# Survival rate by sex
print(df.groupby('sex')['survived'].mean())

# Survival rate by passenger class
print(df.groupby('pclass')['survived'].mean())
```

Notice, for example, females have a much higher survival rate than males. Also, 1st-class passengers survived more often than those in lower classes.

15.1.5 Step 5: Visualizing Data

Visualizations provide intuitive insights.

Survival by Sex and Class

```
sns.barplot(data=df, x='sex', y='survived')
plt.title('Survival Rate by Sex')
plt.show()

sns.barplot(data=df, x='pclass', y='survived')
```

```
plt.title('Survival Rate by Passenger Class')
plt.show()
```

Age Distribution by Survival

Use histograms to compare age distributions:

```
plt.figure(figsize=(10,5))
sns.histplot(data=df, x='age', hue='survived', multiple='stack', bins=30)
plt.title('Age Distribution by Survival')
plt.show()
```

You can see that younger passengers had a higher chance of survival.

Correlation Heatmap

Check correlations between numerical features:

```
corr = df.corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

Note positive correlations between `fare` and `pclass` (inverse relationship due to coding), and a negative correlation between age and survival.

15.1.6 Step 6: Insights and Next Steps

- **Key Insights**: Female passengers and first-class travelers had higher survival chances. Younger passengers also tended to survive more.
- **Missing Data**: Handling missing ages by median is simple but can be improved using more advanced imputation.
- **Feature Engineering**: Adding features like family size (`sibsp + parch`), title extraction from names, or fare bins can enhance predictive modeling.
- **Next Steps**: After EDA, prepare data for machine learning by encoding categorical variables, scaling, and splitting into training/testing sets.

15.1.7 Full Runnable Code Summary

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load data
df = sns.load_dataset('titanic')

# Data overview
```

```python
print(df.info())
print(df.isna().sum())

# Cleaning missing values
df['age'].fillna(df['age'].median(), inplace=True)
df.drop(columns=['deck'], inplace=True)
df['embarked'].fillna(df['embarked'].mode()[0], inplace=True)

# Summary statistics
print(df.describe())
print(df['sex'].value_counts())

# Grouping and aggregation
print(df.groupby('sex')['survived'].mean())
print(df.groupby('pclass')['survived'].mean())

# Visualizations
sns.barplot(data=df, x='sex', y='survived')
plt.title('Survival Rate by Sex')
plt.show()

sns.barplot(data=df, x='pclass', y='survived')
plt.title('Survival Rate by Passenger Class')
plt.show()

plt.figure(figsize=(10,5))
sns.histplot(data=df, x='age', hue='survived', multiple='stack', bins=30)
plt.title('Age Distribution by Survival')
plt.show()

corr = df.corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

15.1.8 Conclusion

This project demonstrated a typical EDA workflow using pandas: loading, cleaning, summarizing, grouping, and visualizing data. Through incremental steps, we uncovered meaningful patterns that guide further modeling or business decisions. With pandas' rich functionality, EDA becomes a smooth and powerful experience — essential for every data scientist.

15.2 Project 2: Time Series Forecasting Preparation

Preparing time series data properly is essential for building accurate forecasting models. This project walks you through the key steps of handling datetime indexes, resampling data, managing missing values, and engineering time-based features, all using a practical stock price dataset. These preprocessing steps ensure your data is clean, consistent, and enriched for modeling.

15.2.1 Step 1: Loading and Inspecting the Data

We'll use historical stock price data for a well-known company, Apple (AAPL), from Yahoo Finance, accessible via `yfinance` or CSV downloads. For this example, let's load a CSV file with daily stock prices.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load dataset (assuming CSV downloaded locally or from URL)
df = pd.read_csv('AAPL.csv', parse_dates=['Date'])
print(df.head())
```

The dataset typically includes columns like `Date`, `Open`, `High`, `Low`, `Close`, `Volume`, and sometimes `Adj Close`.

15.2.2 Step 2: Setting the DateTime Index

Time series analysis requires a datetime index. Set the `Date` column as the DataFrame index and sort it:

```
df.set_index('Date', inplace=True)
df.sort_index(inplace=True)
print(df.index)
```

Having a `DateTimeIndex` enables powerful time-aware operations like resampling and rolling calculations.

15.2.3 Step 3: Handling Missing Data

Real-world time series often have missing dates or data points. For example, stock markets close on weekends, so daily data may miss weekends.

Check for missing dates by creating a full date range and comparing:

```
full_idx = pd.date_range(start=df.index.min(), end=df.index.max(), freq='B')  # 'B' for business days
missing_dates = full_idx.difference(df.index)
print(f"Missing business days: {missing_dates}")
```

To fill missing dates with `NaN`, reindex the DataFrame:

```
df = df.reindex(full_idx)
print(df.head(10))
```

15.2.4 Step 4: Resampling the Data

You may want to change the frequency from daily to weekly or monthly for smoothing or reducing noise.

```
# Resample to monthly frequency, taking mean of 'Close' price
monthly_close = df['Close'].resample('M').mean()
monthly_close.plot(title='Monthly Average Close Price')
plt.show()
```

Or downsample to weekly:

```
weekly_close = df['Close'].resample('W').last()
weekly_close.plot(title='Weekly Close Price')
plt.show()
```

For upsampling (e.g., from monthly to daily), interpolation methods fill gaps:

```
daily_upsampled = monthly_close.resample('D').interpolate()
daily_upsampled.plot(title='Daily Upsampled Close Price (Interpolated)')
plt.show()
```

15.2.5 Step 5: Filling Missing Values

After reindexing or resampling, fill missing values sensibly. Common methods include forward fill (`ffill`) or interpolation:

```
df['Close'].fillna(method='ffill', inplace=True)  # propagate last valid observation forward
df['Close'].fillna(method='bfill', inplace=True)  # or backward fill if needed
```

Avoid dropping missing data blindly, especially in time series where continuity matters.

15.2.6 Step 6: Feature Engineering for Time Series

Adding features based on time can improve forecasting models. Common time features include:

- **Lag features:** previous day's or week's values
- **Rolling statistics:** moving averages, moving standard deviations
- **Date components:** day of week, month, quarter, year

Example: create lag and rolling mean features:

```
df['Close_lag1'] = df['Close'].shift(1)
df['Close_rolling7'] = df['Close'].rolling(window=7).mean()
```

Extract date components:

```
df['day_of_week'] = df.index.dayofweek  # Monday=0, Sunday=6
df['month'] = df.index.month
df['quarter'] = df.index.quarter
df['year'] = df.index.year
```

15.2.7 Step 7: Exploratory Visualization

Visualize the original and engineered features:

```
plt.figure(figsize=(12,6))
plt.plot(df['Close'], label='Close Price')
plt.plot(df['Close_rolling7'], label='7-Day Rolling Mean', linestyle='--')
plt.title('Close Price and 7-Day Rolling Mean')
plt.legend()
plt.show()
```

Plot lagged vs current values to check correlations:

```
plt.scatter(df['Close_lag1'], df['Close'])
plt.xlabel('Close Price (t-1)')
plt.ylabel('Close Price (t)')
plt.title('Lagged Close vs Current Close')
plt.show()
```

15.2.8 Step 8: Summary and Next Steps

- **Datetime Index:** Enables time-aware operations.
- **Resampling:** Flexibly change time granularity.
- **Handling Missing Data:** Use interpolation or filling rather than dropping.
- **Feature Engineering:** Create lag, rolling stats, and date features to capture temporal patterns.
- **Visualization:** Confirm trends and stationarity visually.

This cleaned and enriched dataset is now ready for time series forecasting models like ARIMA, Prophet, or machine learning regressors.

15.2.9 Full Runnable Code Snippet Summary

```
import pandas as pd
import matplotlib.pyplot as plt

# Load and prepare data
df = pd.read_csv('AAPL.csv', parse_dates=['Date'])
df.set_index('Date', inplace=True)
df.sort_index(inplace=True)

# Reindex to business days
full_idx = pd.date_range(df.index.min(), df.index.max(), freq='B')
df = df.reindex(full_idx)

# Fill missing values
df['Close'].fillna(method='ffill', inplace=True)
df['Close'].fillna(method='bfill', inplace=True)

# Resample monthly average close
monthly_close = df['Close'].resample('M').mean()
monthly_close.plot(title='Monthly Average Close Price')
plt.show()

# Feature engineering
df['Close_lag1'] = df['Close'].shift(1)
df['Close_rolling7'] = df['Close'].rolling(window=7).mean()
df['day_of_week'] = df.index.dayofweek
df['month'] = df.index.month

# Visualization
plt.figure(figsize=(12,6))
plt.plot(df['Close'], label='Close Price')
plt.plot(df['Close_rolling7'], label='7-Day Rolling Mean', linestyle='--')
plt.title('Close Price and 7-Day Rolling Mean')
plt.legend()
plt.show()
```

15.3 Project 3: Customer Segmentation with Grouping and Aggregation

Customer segmentation is a key task in marketing analytics, helping businesses tailor campaigns and improve customer engagement by grouping customers based on purchasing behavior. In this project, we'll simulate a purchase dataset and demonstrate how to use pandas' `groupby()`, aggregation, filtering, and sorting functions to segment customers. Finally, we'll visualize the results to extract actionable marketing insights.

15.3.1 Step 1: Simulating Purchase Data

Let's create a sample dataset representing customer purchases over several months. Each record includes customer ID, purchase date, product category, and amount spent.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Seed for reproducibility
np.random.seed(42)

# Create sample data
n_customers = 100
n_purchases = 1000

customer_ids = np.random.choice(range(1, n_customers + 1), size=n_purchases)
dates = pd.date_range(start='2023-01-01', periods=180).to_numpy()
purchase_dates = np.random.choice(dates, size=n_purchases)
categories = np.random.choice(['Electronics', 'Clothing', 'Groceries', 'Books'], size=n_purchases)
amounts = np.round(np.random.exponential(scale=100, size=n_purchases), 2)

# Create DataFrame
df = pd.DataFrame({
    'CustomerID': customer_ids,
    'PurchaseDate': purchase_dates,
    'Category': categories,
    'Amount': amounts
})

print(df.head())
```

15.3.2 Step 2: Creating Summary Metrics via GroupBy and Aggregation

To segment customers, we first summarize their purchasing behavior:

- Total spending
- Number of purchases
- Average purchase amount
- Most frequent purchase category

```
# Group by CustomerID
customer_summary = df.groupby('CustomerID').agg(
    Total_Spent=pd.NamedAgg(column='Amount', aggfunc='sum'),
    Purchase_Count=pd.NamedAgg(column='Amount', aggfunc='count'),
    Avg_Purchase=pd.NamedAgg(column='Amount', aggfunc='mean')
).reset_index()

# Determine each customer's most frequent category
mode_category = df.groupby('CustomerID')['Category'] \
                  .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else np.nan)

customer_summary['Top_Category'] = mode_category.values
```

```python
print(customer_summary.head())
```

15.3.3 Step 3: Segmenting Customers

A simple segmentation approach divides customers based on spending and purchase frequency:

- **High Value**: Total spent > $500
- **Frequent Buyers**: Purchase count > 10
- **Occasional**: Otherwise

```python
def segment_customer(row):
    if row['Total_Spent'] > 500:
        return 'High Value'
    elif row['Purchase_Count'] > 10:
        return 'Frequent Buyer'
    else:
        return 'Occasional'

customer_summary['Segment'] = customer_summary.apply(segment_customer, axis=1)
print(customer_summary['Segment'].value_counts())
```

15.3.4 Step 4: Filtering and Sorting Segments

Identify the top 5 high-value customers by total spend:

```python
top_high_value = customer_summary[customer_summary['Segment'] == 'High Value'] \
    .sort_values(by='Total_Spent', ascending=False) \
    .head(5)

print("Top 5 High-Value Customers:")
print(top_high_value[['CustomerID', 'Total_Spent', 'Purchase_Count', 'Top_Category']])
```

15.3.5 Step 5: Visualizing Customer Segments

Visualizations help communicate segment characteristics:

- Bar plot of segment counts
- Boxplot of spending by segment

```python
plt.figure(figsize=(10,4))

# Bar plot of segment sizes
plt.subplot(1, 2, 1)
customer_summary['Segment'].value_counts().plot(kind='bar', color=['green', 'blue', 'orange'])
plt.title('Customer Segment Counts')
plt.ylabel('Number of Customers')

# Boxplot of total spent by segment
plt.subplot(1, 2, 2)
customer_summary.boxplot(column='Total_Spent', by='Segment', grid=False)
plt.title('Total Spending by Segment')
plt.suptitle('')  # Remove automatic title
plt.ylabel('Total Spent ($)')

plt.tight_layout()
plt.show()
```

15.3.6 Step 6: Interpretation and Marketing Insights

- **High Value** customers contribute a large share of revenue but may be fewer in number.
- **Frequent Buyers** show loyalty and regular engagement, valuable for subscription or upselling strategies.
- **Occasional** customers might need targeted promotions or incentives to increase frequency.

Analyzing the top categories per segment can tailor campaigns. For example, if **High Value** customers mostly buy electronics, personalized offers on new gadgets may increase retention.

15.3.7 Full Runnable Code Summary

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

n_customers = 100
n_purchases = 1000
customer_ids = np.random.choice(range(1, n_customers + 1), size=n_purchases)
dates = pd.date_range(start='2023-01-01', periods=180).to_numpy()
purchase_dates = np.random.choice(dates, size=n_purchases)
categories = np.random.choice(['Electronics', 'Clothing', 'Groceries', 'Books'], size=n_purchases)
amounts = np.round(np.random.exponential(scale=100, size=n_purchases), 2)

df = pd.DataFrame({
    'CustomerID': customer_ids,
    'PurchaseDate': purchase_dates,
    'Category': categories,
    'Amount': amounts
})

# Summarize customer purchase behavior
customer_summary = df.groupby('CustomerID').agg(
    Total_Spent=pd.NamedAgg(column='Amount', aggfunc='sum'),
    Purchase_Count=pd.NamedAgg(column='Amount', aggfunc='count'),
    Avg_Purchase=pd.NamedAgg(column='Amount', aggfunc='mean')
).reset_index()

mode_category = df.groupby('CustomerID')['Category'] \
                  .agg(lambda x: x.mode().iloc[0] if not x.mode().empty else np.nan)
customer_summary['Top_Category'] = mode_category.values

# Define segments
def segment_customer(row):
    if row['Total_Spent'] > 500:
        return 'High Value'
    elif row['Purchase_Count'] > 10:
        return 'Frequent Buyer'
    else:
        return 'Occasional'

customer_summary['Segment'] = customer_summary.apply(segment_customer, axis=1)

# Top 5 High Value customers
top_high_value = customer_summary[customer_summary['Segment'] == 'High Value'] \
    .sort_values(by='Total_Spent', ascending=False).head(5)

print(top_high_value[['CustomerID', 'Total_Spent', 'Purchase_Count', 'Top_Category']])

# Visualization
plt.figure(figsize=(10,4))
plt.subplot(1, 2, 1)
customer_summary['Segment'].value_counts().plot(kind='bar', color=['green', 'blue', 'orange'])
plt.title('Customer Segment Counts')
plt.ylabel('Number of Customers')

plt.subplot(1, 2, 2)
customer_summary.boxplot(column='Total_Spent', by='Segment', grid=False)
plt.title('Total Spending by Segment')
plt.suptitle('')
plt.ylabel('Total Spent ($)')
plt.tight_layout()
plt.show()
```

15.3.8 Conclusion

This project demonstrates how pandas' powerful grouping and aggregation tools enable meaningful customer segmentation. By summarizing purchase behaviors and adding domain-specific segmentation logic, businesses can discover actionable insights and improve marketing effectiveness. Visualizations further support data-driven decisions and communication with stakeholders.

Ready for Project 4 on data cleaning and merging?

15.4 Project 4: Data Cleaning and Merging from Multiple Sources

In real-world data science projects, data rarely comes from a single, clean source. Instead, datasets originate from multiple formats and systems — CSV files, databases, APIs — each with inconsistencies, missing values, or overlapping information. This project walks through combining such disparate data sources, cleaning them, handling missing values, and producing a unified dataset ready for analysis.

15.4.1 Step 1: Loading Multiple Datasets

Suppose we have customer purchase data from two different branches of a store, each stored as a CSV file. Additionally, we have a product catalog stored in JSON format.

```python
import pandas as pd
import numpy as np

# Load sales data from two CSVs
branch1 = pd.read_csv('branch1_sales.csv')
branch2 = pd.read_csv('branch2_sales.csv')

# Load product catalog from JSON
products = pd.read_json('products.json')

# Show sample data
print(branch1.head())
print(branch2.head())
print(products.head())
```

15.4.2 Step 2: Inspecting and Cleaning Individual DataFrames

Before merging, inspect the data for inconsistencies:

- Check column names and data types
- Identify missing values
- Standardize formats (e.g., date formats, string capitalization)

```python
print(branch1.info())
print(branch2.info())

# Convert purchase dates to datetime
branch1['PurchaseDate'] = pd.to_datetime(branch1['PurchaseDate'], errors='coerce')
branch2['PurchaseDate'] = pd.to_datetime(branch2['PurchaseDate'], errors='coerce')

# Normalize product IDs and customer IDs to string for consistency
branch1['ProductID'] = branch1['ProductID'].astype(str)
branch2['ProductID'] = branch2['ProductID'].astype(str)
```

```
branch1['CustomerID'] = branch1['CustomerID'].astype(str)
branch2['CustomerID'] = branch2['CustomerID'].astype(str)

# Standardize string columns (e.g., product names)
products['ProductName'] = products['ProductName'].str.strip().str.title()
```

15.4.3 Step 3: Combining Sales Data from Branches

Use `pd.concat()` to stack branch data vertically since they share the same schema.

```
sales = pd.concat([branch1, branch2], ignore_index=True)
print(f"Combined sales records: {sales.shape[0]}")
```

15.4.4 Step 4: Handling Duplicates and Missing Data

After concatenation, some transactions might be duplicated due to overlapping data or data entry errors.

```
# Detect duplicates based on key columns
duplicates = sales.duplicated(subset=['TransactionID'])
print(f"Number of duplicate transactions: {duplicates.sum()}")

# Remove duplicates, keeping the first occurrence
sales = sales.drop_duplicates(subset=['TransactionID'])
```

Missing data is also common; handle it based on context:

```
# Check for missing values
print(sales.isnull().sum())

# For missing purchase dates, drop rows or fill if possible
sales = sales.dropna(subset=['PurchaseDate'])

# For missing amounts, fill with zero or median depending on business logic
sales['Amount'] = sales['Amount'].fillna(sales['Amount'].median())
```

15.4.5 Step 5: Merging Sales with Product Catalog

To enrich sales data with product details, perform a merge on the `ProductID` key.

```
# Merge sales with products on ProductID
full_data = sales.merge(products, on='ProductID', how='left', indicator=True)

# Check for unmatched products
unmatched = full_data[full_data['_merge'] == 'left_only']
print(f"Unmatched product IDs: {unmatched['ProductID'].nunique()}")
```

```
# Depending on analysis needs, handle unmatched products:
# Option 1: Drop unmatched records
full_data = full_data[full_data['_merge'] == 'both']

# Option 2: Investigate and fix product catalog or sales data errors
```

15.4.6 Step 6: Validating and Finalizing the Dataset

Validation ensures data quality before analysis:

- Confirm no missing critical fields (e.g., customer ID, amount)
- Validate data ranges (e.g., amounts > 0)
- Check data types

```
# Drop rows with missing customer IDs
full_data = full_data.dropna(subset=['CustomerID'])

# Remove records with negative or zero amounts
full_data = full_data[full_data['Amount'] > 0]

# Final info
print(full_data.info())
```

15.4.7 Step 7: Summary and Export

After cleaning and merging, save the unified dataset for downstream analysis:

```
full_data.to_csv('cleaned_sales_data.csv', index=False)
```

15.4.8 Recap and Best Practices

- **Consistent formats:** Normalize IDs and datetime fields early.
- **Concatenate similar data:** Use `concat()` to stack datasets with same schema.
- **Merge to enrich:** Use `merge()` for combining on keys; choose join types (`inner`, `left`) carefully.
- **Handle duplicates:** Use `duplicated()` and `drop_duplicates()` to remove repeats.
- **Address missing data:** Tailor filling or dropping strategies based on domain knowledge.
- **Validate post-merge:** Check for mismatches and data integrity.
- **Document transformations:** Keep track of cleaning steps for reproducibility.

15.4.9 Full Code Example (Simulated Data)

Below is a runnable minimal example using simulated data mimicking the above workflow:

```python
import pandas as pd
import numpy as np

# Simulate branch sales
branch1 = pd.DataFrame({
    'TransactionID': ['T1', 'T2', 'T3'],
    'CustomerID': ['C1', 'C2', 'C3'],
    'ProductID': ['P1', 'P2', 'P3'],
    'PurchaseDate': ['2023-01-01', '2023-01-02', '2023-01-03'],
    'Amount': [100, 150, 200]
})

branch2 = pd.DataFrame({
    'TransactionID': ['T4', 'T2', 'T5'],  # Notice T2 duplicate
    'CustomerID': ['C4', 'C2', 'C5'],
    'ProductID': ['P1', 'P2', 'P4'],
    'PurchaseDate': ['2023-01-04', '2023-01-02', None],
    'Amount': [120, None, 180]
})

products = pd.DataFrame({
    'ProductID': ['P1', 'P2', 'P3'],
    'ProductName': ['Widget', 'Gadget', 'Doohickey'],
    'Category': ['Tools', 'Electronics', 'Misc']
})

# Standardize dates
for df in [branch1, branch2]:
    df['PurchaseDate'] = pd.to_datetime(df['PurchaseDate'], errors='coerce')

# Concatenate sales
sales = pd.concat([branch1, branch2], ignore_index=True)

# Drop duplicates by TransactionID
sales = sales.drop_duplicates(subset=['TransactionID'])

# Handle missing dates and amounts
sales = sales.dropna(subset=['PurchaseDate'])
sales['Amount'] = sales['Amount'].fillna(sales['Amount'].median())

# Merge with products
full_data = sales.merge(products, on='ProductID', how='left')

print(full_data)
```

15.4.10 Conclusion

Combining and cleaning data from multiple sources is crucial for robust data analysis. This project demonstrated core pandas techniques to unify, clean, and validate data, preparing it for deeper insights or modeling. Mastering these skills ensures you can handle real-world messy data with confidence and accuracy.

Ready to explore the final project—an end-to-end data science pipeline using pandas?

15.5 Project 5: End-to-End Data Science Pipeline using Pandas

In this final project, we'll bring together all the concepts covered throughout this book into a comprehensive, reproducible data science pipeline using pandas. This includes data loading, cleaning, transformation, analysis, and exporting results. The goal is to demonstrate how to structure your code, handle real-world challenges, and produce meaningful insights in a clear, maintainable way.

15.5.1 Dataset Overview

We will use a publicly available **retail sales dataset** (simulated for this example) that includes transaction records, customer info, and product details. This dataset features multiple sources and formats, missing values, and time series data. The pipeline will:

- Load data from CSV and JSON files
- Clean and preprocess the data (handle missing data, type conversions)
- Perform feature engineering (date features, categorical encoding)
- Aggregate and analyze customer sales by segment
- Export cleaned and aggregated data for reporting

15.5.2 Step 1: Setup and Data Loading

```
import pandas as pd
import numpy as np

# Load datasets
sales = pd.read_csv('sales.csv')  # transaction data
customers = pd.read_json('customers.json')  # customer profiles
products = pd.read_csv('products.csv')  # product catalog

# Preview data
print(sales.head())
print(customers.head())
print(products.head())
```

15.5.3 Step 2: Data Cleaning and Preprocessing

Handle missing values, standardize formats, and ensure data types are correct.

```python
# Convert dates
sales['PurchaseDate'] = pd.to_datetime(sales['PurchaseDate'], errors='coerce')

# Drop sales records with missing PurchaseDate or Amount
sales = sales.dropna(subset=['PurchaseDate', 'Amount'])

# Fill missing customer IDs (if any) with placeholder
sales['CustomerID'] = sales['CustomerID'].fillna('Unknown').astype(str)
customers['CustomerID'] = customers['CustomerID'].astype(str)

# Normalize string columns
products['ProductName'] = products['ProductName'].str.strip().str.title()
customers['CustomerName'] = customers['CustomerName'].str.strip().str.title()

# Ensure IDs are strings
sales['ProductID'] = sales['ProductID'].astype(str)
products['ProductID'] = products['ProductID'].astype(str)
```

15.5.4 Step 3: Merging Datasets

Combine sales with customer and product information to enrich the dataset.

```python
# Merge sales with products
sales_products = sales.merge(products, on='ProductID', how='left')

# Merge with customer data
full_data = sales_products.merge(customers, on='CustomerID', how='left')

print(full_data.info())
```

15.5.5 Step 4: Feature Engineering

Create new features useful for analysis:

- Extract year, month, weekday from purchase date
- Convert categorical fields to pandas `category` dtype to save memory and speed up grouping

```python
full_data['Year'] = full_data['PurchaseDate'].dt.year
full_data['Month'] = full_data['PurchaseDate'].dt.month
full_data['Weekday'] = full_data['PurchaseDate'].dt.day_name()

# Convert categories
for col in ['Category', 'CustomerSegment', 'Weekday']:
    if col in full_data.columns:
        full_data[col] = full_data[col].astype('category')
```

```
print(full_data.head())
```

15.5.6 Step 5: Aggregation and Customer Segmentation

Group customers by segment and analyze their total spending and transaction counts.

```
# Group sales by customer segment and month
segment_summary = (full_data
                   .groupby(['CustomerSegment', 'Year', 'Month'])
                   .agg(TotalSales=('Amount', 'sum'),
                        TransactionCount=('TransactionID', 'count'))
                   .reset_index())

print(segment_summary.head())
```

15.5.7 Step 6: Visualization (Optional, requires matplotlib)

Quickly plot sales trends by customer segment.

```
import matplotlib.pyplot as plt

for segment in segment_summary['CustomerSegment'].unique():
    subset = segment_summary[segment_summary['CustomerSegment'] == segment]
    plt.plot(pd.to_datetime(subset[['Year', 'Month']].assign(DAY=1)),
             subset['TotalSales'], label=segment)

plt.title('Monthly Sales by Customer Segment')
plt.xlabel('Month')
plt.ylabel('Total Sales')
plt.legend()
plt.show()
```

15.5.8 Step 7: Export Cleaned and Aggregated Data

Save processed datasets for reporting or further modeling.

```
full_data.to_parquet('cleaned_retail_data.parquet')
segment_summary.to_csv('customer_segment_summary.csv', index=False)
```

15.5.9 Step 8: Wrap as Functions for Reproducibility

Organizing code in functions makes the pipeline modular and easy to maintain.

```python
def load_data():
    sales = pd.read_csv('sales.csv')
    customers = pd.read_json('customers.json')
    products = pd.read_csv('products.csv')
    return sales, customers, products

def clean_data(sales, customers, products):
    sales['PurchaseDate'] = pd.to_datetime(sales['PurchaseDate'], errors='coerce')
    sales = sales.dropna(subset=['PurchaseDate', 'Amount'])
    sales['CustomerID'] = sales['CustomerID'].fillna('Unknown').astype(str)
    customers['CustomerID'] = customers['CustomerID'].astype(str)
    products['ProductName'] = products['ProductName'].str.strip().str.title()
    sales['ProductID'] = sales['ProductID'].astype(str)
    products['ProductID'] = products['ProductID'].astype(str)
    return sales, customers, products

def merge_data(sales, customers, products):
    sales_products = sales.merge(products, on='ProductID', how='left')
    full_data = sales_products.merge(customers, on='CustomerID', how='left')
    return full_data

def feature_engineering(full_data):
    full_data['Year'] = full_data['PurchaseDate'].dt.year
    full_data['Month'] = full_data['PurchaseDate'].dt.month
    full_data['Weekday'] = full_data['PurchaseDate'].dt.day_name()
    for col in ['Category', 'CustomerSegment', 'Weekday']:
        if col in full_data.columns:
            full_data[col] = full_data[col].astype('category')
    return full_data

def aggregate_data(full_data):
    return (full_data.groupby(['CustomerSegment', 'Year', 'Month'])
                    .agg(TotalSales=('Amount', 'sum'),
                         TransactionCount=('TransactionID', 'count'))
                    .reset_index())

def run_pipeline():
    sales, customers, products = load_data()
    sales, customers, products = clean_data(sales, customers, products)
    full_data = merge_data(sales, customers, products)
    full_data = feature_engineering(full_data)
    segment_summary = aggregate_data(full_data)
    full_data.to_parquet('cleaned_retail_data.parquet')
    segment_summary.to_csv('customer_segment_summary.csv', index=False)
    print("Pipeline complete. Files saved.")

if __name__ == "__main__":
    run_pipeline()
```

15.5.10 Final Thoughts

This end-to-end pipeline demonstrates:

- **Data loading** from various formats

- **Cleaning and preprocessing** including date parsing and missing value handling
- **Merging datasets** to enrich the base data
- **Feature engineering** with datetime and categorical data
- **Aggregation and segmentation** to extract insights
- **Exporting results** for sharing or further analysis
- **Modular, reusable code** structure

Using pandas effectively in such pipelines boosts productivity, ensures reproducibility, and prepares data science practitioners for real-world challenges. The techniques demonstrated here form the foundation for more advanced projects, including machine learning and real-time analytics.

Feel free to adapt and expand this pipeline with domain-specific transformations, richer visualizations, or integration with external APIs and databases for full-scale production workflows.