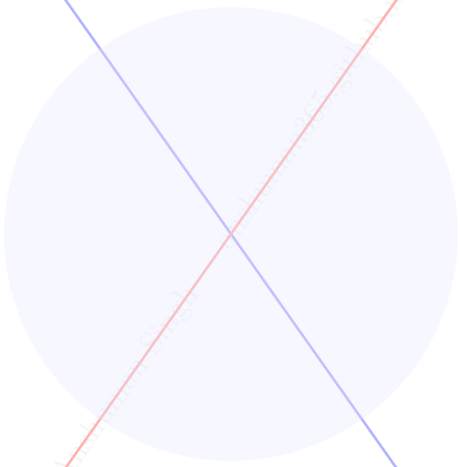
Pandas Mastery

From Beginner to Advanced

A Comprehensive Guide to Data Analysis with Python

Anshuman Singh



November 27, 2025

First Edition: November 27, 2025

ISBN: 978-0-000000-00-0

Cover Design: Data Science Edition

Typeset in LaTeX

Contents

List of Figures

Part I

Getting Started with Pandas

Chapter 1

Introduction to Pandas

1.1 What is Pandas?

Pandas is a fast, powerful, flexible, and easy-to-use open-source data analysis and manipulation tool built on top of the Python programming language. The name ”Pandas” is derived from the term ”panel data”, an econometrics term for multidimensional structured data sets.

1.1.1 Why Use Pandas?

Pandas has become the de facto standard for data manipulation in Python due to its intuitive data structures and powerful data analysis capabilities. It bridges the gap between spreadsheet applications and programming languages, providing both ease of use and programmatic power.

Key features of Pandas include:

- **DataFrame object**: For data manipulation with integrated indexing
- **Data I/O**: Tools for reading and writing data between in-memory data structures and different file formats (CSV, Excel, JSON, SQL, etc.)
- **Data alignment**: Integrated handling of missing data with intelligent data alignment
- **Reshaping and pivoting**: Powerful tools for reshaping and pivoting data sets
- **Label-based slicing**: Sophisticated indexing and subsetting of large data sets
- **Group by operations**: Split-apply-combine operations on data sets
- **High performance merging**: Efficient merging and joining of data
- **Time series functionality**: Date range generation and frequency conversion, moving window statistics, date shifting and lagging

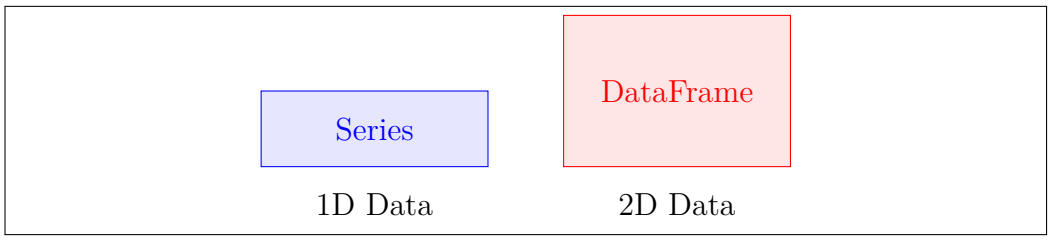


Figure 1.1: Pandas data structures: Series (1-dimensional) and DataFrame (2-dimensional)

1.2 Installing Pandas

Pandas can be installed using various package managers. Here are the most common methods:

```
# Using pip (most common)
pip install pandas

# Using conda (for Anaconda/Miniconda users)
conda install pandas

# For specific version
pip install pandas==2.0.0

# Installing with optional dependencies
pip install pandas[performance]

# From source (advanced users)
git clone https://github.com/pandas-dev/pandas.git
cd pandas
python setup.py install
```

Listing 1.1: Installing Pandas using different methods

1.2.1 Verifying Installation

After installation, verify that pandas is properly installed:

```
import pandas as pd
print(f"Pandas version: {pd.__version__}")
print(f"Pandas location: {pd.__file__}")
```

Listing 1.2: Verifying pandas installation

1.3 Importing Pandas

The conventional way to import pandas in Python follows these best practices:

```
import pandas as pd
import numpy as np  # Often used together with pandas

# Common aliases for data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Display settings for better output
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
```

Listing 1.3: Importing pandas and related libraries

1.3.1 Why the pd alias?

The `pd` alias has become a community standard for several reasons:

- Shorter and more convenient to type
- Consistent with community conventions

- Avoids namespace conflicts
- Makes code more readable and professional

1.4 Pandas Ecosystem

Pandas integrates seamlessly with other Python libraries:

- **NumPy**: Foundation for pandas data structures
- **Matplotlib/Seaborn**: For data visualization
- **Scikit-learn**: For machine learning
- **Statsmodels**: For statistical analysis
- **SQLAlchemy**: For database operations

1.5 Exercises

1. Install pandas on your system using your preferred method and verify the installation.
2. Create a Python script that imports pandas and prints the version information.
3. What are the main advantages of using pandas over plain Python lists or NumPy arrays for data analysis?
4. List three common file formats that pandas can read and write.

1.6 Solutions

1.
```python
# Verify installation and version
import pandas as pd
print(f"Pandas version: {pd.__version__}")

# Check basic functionality
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
print(df)
```

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

print(f"Pandas version: {pd.__version__}")
print(f"NumPy version: {np.__version__}")

# Additional system information
import sys
print(f"Python version: {sys.version}")
```

3. Main advantages of pandas:
    - Label-based indexing for intuitive data access

- Built-in handling of missing data
- Powerful group by operations
- Integrated time series functionality
- Easy data I/O with multiple file formats
- Memory efficiency with optimized data structures

4. Three common file formats pandas supports:

- CSV (Comma Separated Values)
- Excel (.xlsx, .xls)
- JSON (JavaScript Object Notation)
- SQL databases
- Parquet
- HDF5

Chapter 2

Series and DataFrames: Core Data Structures

2.1 Understanding Series

A Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). Think of it as a column in a spreadsheet or a single variable in statistics.

2.1.1 Creating Series Objects

```
import pandas as pd
import numpy as np

# Creating Series from list (most common)
s1 = pd.Series([1, 3, 5, np.nan, 6, 8])
print("Series from list:")
print(s1)

# Creating Series with custom index
s2 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print("\nSeries with custom index:")
print(s2)

# Creating Series from dictionary
s3 = pd.Series({'a': 1, 'b': 2, 'c': 3})
print("\nSeries from dictionary:")
print(s3)

# Creating Series with specified data type
s4 = pd.Series([1, 2, 3], dtype=np.float64)
print("\nSeries with specified dtype:")
print(s4)

# Creating Series from NumPy array
arr = np.array([1, 2, 3, 4])
s5 = pd.Series(arr)
print("\nSeries from NumPy array:")
print(s5)
```

Listing 2.1: Various ways to create Series

2.1.2 Series Attributes and Methods

```
# Create a sample series
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

# Basic attributes
```

```
print(f"Values: {s.values}")
print(f"Index: {s.index}")
print(f"Data type: {s.dtype}")
print(f"Shape: {s.shape}")
print(f"Size: {s.size}")
print(f"Name: {s.name}")

# Basic methods
print(f"Sum: {s.sum()}")
print(f"Mean: {s.mean()}")
print(f"Standard deviation: {s.std()}")
print(f"Minimum: {s.min()}")
print(f"Maximum: {s.max()}")

# Boolean operations
print(f"Greater than 25: {s > 25}")
```

Listing 2.2: Working with Series attributes

2.2 Understanding DataFrames

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types, similar to a spreadsheet or SQL table. It's the most commonly used pandas object.

2.2.1 Creating DataFrames

```
# Creating DataFrame from dictionary (most common)
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'City': ['New York', 'London', 'Tokyo', 'Paris'],
    'Salary': [50000, 60000, 70000, 80000]
}
df = pd.DataFrame(data)
print("DataFrame from dictionary:")
print(df)

# Creating DataFrame with custom index
df_indexed = pd.DataFrame(data, index=['EMP001', 'EMP002', 'EMP003', 'EMP004'])
print("\nDataFrame with custom index:")
print(df_indexed)

# Creating DataFrame from list of lists
data_list = [
    ['Alice', 25, 'New York', 50000],
    ['Bob', 30, 'London', 60000],
    ['Charlie', 35, 'Tokyo', 70000],
    ['David', 40, 'Paris', 80000]
]
columns = ['Name', 'Age', 'City', 'Salary']
df_from_list = pd.DataFrame(data_list, columns=columns)
print("\nDataFrame from list of lists:")
print(df_from_list)

# Creating DataFrame from NumPy array
np_array = np.random.rand(4, 3)
df_from_np = pd.DataFrame(np_array, columns=['A', 'B', 'C'])
```

```
print("\nDataFrame from NumPy array:")
print(df_from_np)
```

Listing 2.3: Various ways to create DataFrames

2.3 Basic DataFrame Operations

```
# Create sample DataFrame
data = {
    'Product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor', 'Headphones'],
    'Price': [1000, 25, 75, 300, 150],
    'Quantity': [5, 20, 15, 8, 12],
    'Category': ['Electronics', 'Accessories', 'Accessories', 'Electronics', '
       Audio']
}
df = pd.DataFrame(data)

# Display basic information
print("DataFrame head:")
print(df.head())

print("\nDataFrame tail:")
print(df.tail(2))

print("\nDataFrame info:")
print(df.info())

print("\nDataFrame description:")
print(df.describe())

# Shape and dimensions
print(f"\nShape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")
print(f"Index: {df.index.tolist()}")

# Access columns
print("\nSingle column:")
print(df['Product'])

print("\nMultiple columns:")
print(df[['Product', 'Price']])

# Using dot notation (when column names are valid Python identifiers)
print("\nUsing dot notation:")
print(df.Price)
```

Listing 2.4: Essential DataFrame operations

2.4 DataFrame Attributes and Metadata

```
# DataFrame attributes
print(f"Columns: {df.columns}")
print(f"Index: {df.index}")
print(f"Shape: {df.shape}")
print(f"Data types:\n{df.dtypes}")
print(f"Memory usage: {df.memory_usage(deep=True).sum()} bytes")
```

```python

# Transposing DataFrame
print("\nTransposed DataFrame:")
print(df.T)

# Converting to other formats
print("\nAs NumPy array:")
print(df.values)

print("\nAs dictionary:")
print(df.to_dict())
```

Listing 2.5: Working with DataFrame metadata

2.5 Exercises

1. Create a Series that represents daily temperatures for a week (Monday to Sunday) with appropriate values.

2. Create a DataFrame containing information about 5 books with columns: Title, Author, Year, Pages, and Price.

3. From your books DataFrame, extract only the Title and Price columns.

4. Calculate the average price of books in your DataFrame.

5. Add a new column 'Rating' to your books DataFrame with random ratings between 1 and 5.

2.6 Solutions

1.
```python
temperatures = pd.Series(
    [22, 24, 23, 25, 26, 27, 26],
    index=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', '
       Saturday', 'Sunday']
)
print(temperatures)
```

```python
books_data = {
    'Title': ['Python Basics', 'Data Science Guide', 'ML Advanced', 'Deep
       Learning', 'Pandas Cookbook'],
    'Author': ['Author A', 'Author B', 'Author C', 'Author D', 'Author E'],
    'Year': [2020, 2021, 2022, 2023, 2024],
    'Pages': [300, 450, 520, 600, 350],
    'Price': [29.99, 39.99, 49.99, 59.99, 34.99]
}
books_df = pd.DataFrame(books_data)
print(books_df)
```

3.
```python
title_price = books_df[['Title', 'Price']]
print(title_price)
```

```
average_price = books_df['Price'].mean()
print(f"Average price: ${average_price:.2f}")
```

4.

```
import random
books_df['Rating'] = [random.randint(1, 5) for _ in range(len(books_df))]
print(books_df)
```

Chapter 3

Indexing, Slicing, and Data Selection

3.1 Understanding Indexing Methods

Pandas provides several methods for selecting and slicing data. Understanding these methods is crucial for efficient data manipulation.

3.1.1 loc[] - Label-based Indexing

```
# Create sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [25, 30, 35, 40, 28],
    'City': ['NY', 'London', 'Tokyo', 'Paris', 'Berlin'],
    'Salary': [70000, 80000, 90000, 100000, 75000]
}
df = pd.DataFrame(data)
df.index = ['A', 'B', 'C', 'D', 'E']  # Custom index

# Single row by label
print("Row A:")
print(df.loc['A'])

# Multiple rows by labels
print("\nRows A and C:")
print(df.loc[['A', 'C']])

# Rows and specific columns
print("\nRows A-C, Name and Salary columns:")
print(df.loc['A':'C', ['Name', 'Salary']])

# Boolean indexing with loc
print("\nPeople with Salary > 80000:")
print(df.loc[df['Salary'] > 80000])

# Conditional selection on specific columns
print("\nNames of people in London:")
print(df.loc[df['City'] == 'London', 'Name'])
```

Listing 3.1: Using loc for label-based indexing

3.1.2 iloc[] - Integer-based Indexing

```
# Integer-based indexing
print("First row:")
print(df.iloc[0])

print("\nFirst three rows:")
print(df.iloc[0:3])

```

```
print("\nSpecific rows and columns by position:")
print(df.iloc[[0, 2, 4], [0, 2]])  # Rows 0,2,4 and columns 0,2

print("\nLast two rows:")
print(df.iloc[-2:])

# Using with conditions (less common but possible)
print("\nUsing iloc with conditions:")
print(df.iloc[(df['Age'] > 30).values])
```

Listing 3.2: Using iloc for integer-based indexing

3.1.3 Boolean Indexing

```
# Single condition
young_employees = df[df['Age'] < 30]
print("Employees younger than 30:")
print(young_employees)

# Multiple conditions using & (and), | (or)
high_salary_young = df[(df['Salary'] > 80000) & (df['Age'] < 40)]
print("\nYoung employees with high salary:")
print(high_salary_young)

# Using isin() for multiple values
cities = ['London', 'Paris']
in_europe = df[df['City'].isin(cities)]
print("\nEmployees in European cities:")
print(in_europe)

# Using str methods for string filtering
names_with_a = df[df['Name'].str.contains('a', case=False)]
print("\nNames containing 'a':")
print(names_with_a)

# Using query() method (alternative syntax)
high_paid = df.query('Salary > 85000 and Age > 30')
print("\nHigh paid experienced employees:")
print(high_paid)
```

Listing 3.3: Boolean indexing techniques

3.2 Setting Values with Indexing

```
# Setting single value
df.loc['A', 'Salary'] = 75000

# Setting multiple values
df.loc['B':'D', 'Salary'] = [85000, 95000, 105000]

# Conditional setting
df.loc[df['City'] == 'London', 'Salary'] = 90000

# Using mask to set values
df['Bonus'] = 0
df.loc[df['Salary'] > 90000, 'Bonus'] = 10000
```

```
13
14 print("Updated DataFrame:")
15 print(df)
```

Listing 3.4: Modifying data using indexing

3.3 Exercises

1. Create a DataFrame with 10 rows of sample sales data including: Product, Category, Price, Quantity, Date.
2. Use loc to select all products in a specific category with price greater than 50.
3. Use iloc to select the first 5 rows and only the Product and Price columns.
4. Use boolean indexing to find products with quantity less than 10.
5. Add a new column 'Total' that calculates Price * Quantity for each row.

3.4 Solutions

1.
```
import pandas as pd
import numpy as np

np.random.seed(42)
sales_data = {
    'Product': [f'Product_{i}' for i in range(1, 11)],
    'Category': np.random.choice(['Electronics', 'Clothing', 'Books', 'Home
        '], 10),
    'Price': np.random.randint(10, 200, 10),
    'Quantity': np.random.randint(1, 20, 10),
    'Date': pd.date_range('2024-01-01', periods=10)
}
sales_df = pd.DataFrame(sales_data)
print(sales_df)
```

```
electronics_expensive = sales_df.loc[
    (sales_df['Category'] == 'Electronics') &
    (sales_df['Price'] > 50)
]
print(electronics_expensive)
```

3.
```
first_five = sales_df.iloc[0:5, [0, 2]]  # Rows 0-4, columns 0 and 2
print(first_five)
```

```
low_quantity = sales_df[sales_df['Quantity'] < 10]
print(low_quantity)
```

5.
```
sales_df['Total'] = sales_df['Price'] * sales_df['Quantity']
print(sales_df)
```

Chapter 4

Data Cleaning and Preparation

4.1 Handling Missing Data

4.1.1 Detecting Missing Values

```
# Create DataFrame with missing values
data = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, np.nan, 5],
    'C': [1, 2, 3, 4, 5],
    'D': [np.nan, np.nan, np.nan, np.nan, np.nan]
}
df = pd.DataFrame(data)

print("DataFrame with missing values:")
print(df)

print("\nMissing values per column:")
print(df.isnull().sum())

print("\nPercentage of missing values:")
print(df.isnull().sum() / len(df) * 100)

print("\nRows with any missing values:")
print(df[df.isnull().any(axis=1)])

print("\nColumns with any missing values:")
print(df.columns[df.isnull().any()].tolist())
```

Listing 4.1: Identifying missing data

4.1.2 Removing Missing Data

```
# Remove rows with any missing values
df_dropped_rows = df.dropna()
print("After dropping rows with any NaN:")
print(df_dropped_rows)

# Remove columns with any missing values
df_dropped_cols = df.dropna(axis=1)
print("\nAfter dropping columns with any NaN:")
print(df_dropped_cols)

# Remove rows where all values are NaN
df_dropped_all_na = df.dropna(how='all')
print("\nAfter dropping rows where all values are NaN:")
print(df_dropped_all_na)

# Remove rows with less than 3 non-NaN values
```

```
df_dropped_thresh = df.dropna(thresh=3)
print("\nAfter dropping rows with less than 3 non-NaN values:")
print(df_dropped_thresh)
```

Listing 4.2: Removing missing values

4.1.3 Filling Missing Data

```
# Fill with specific value
df_filled_0 = df.fillna(0)
print("Filled with 0:")
print(df_filled_0)

# Fill with mean of each column
df_filled_mean = df.fillna(df.mean())
print("\nFilled with column means:")
print(df_filled_mean)

# Forward fill
df_ffill = df.fillna(method='ffill')
print("\nForward filled:")
print(df_ffill)

# Backward fill
df_bfill = df.fillna(method='bfill')
print("\nBackward filled:")
print(df_bfill)

# Different fill values for different columns
fill_values = {'A': 0, 'B': df['B'].mean(), 'C': 999}
df_custom_fill = df.fillna(fill_values)
print("\nCustom filled:")
print(df_custom_fill)

# Using interpolate for time series
df_interpolated = df.interpolate()
print("\nInterpolated:")
print(df_interpolated)
```

Listing 4.3: Filling missing values

4.2 Data Type Conversion

```
# Create sample data with mixed types
data = {
    'integers_as_strings': ['1', '2', '3', '4', '5'],
    'floats_as_strings': ['1.1', '2.2', '3.3', '4.4', '5.5'],
    'booleans_as_strings': ['True', 'False', 'True', 'False', 'True'],
    'dates_as_strings': ['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04',
        '2024-01-05'],
    'mixed_numeric': [1, '2', 3.0, '4', 5]
}
df_types = pd.DataFrame(data)

print("Original data types:")
print(df_types.dtypes)
```

```

# Convert to appropriate types
df_types['integers_as_strings'] = df_types['integers_as_strings'].astype(int)
df_types['floats_as_strings'] = df_types['floats_as_strings'].astype(float)
df_types['booleans_as_strings'] = df_types['booleans_as_strings'].map({'True':
    True, 'False': False})
df_types['dates_as_strings'] = pd.to_datetime(df_types['dates_as_strings'])
df_types['mixed_numeric'] = pd.to_numeric(df_types['mixed_numeric'], errors='
    coerce')

print("\nAfter conversion:")
print(df_types.dtypes)
print(df_types)
```

Listing 4.4: Converting data types

4.3 String Operations

```
# Create sample text data
text_data = {
    'name': ['john doe', 'JANE SMITH', 'Bob Johnson', 'alice brown', 'CHARLIE
       WILSON'],
    'email': ['john@email.com', 'jane@test.org', 'bob@company.com', 'alice@home.
       net', 'charlie@work.io'],
    'description': ['Python developer with 5 years experience',
                    'Data scientist specializing in ML',
                    'Senior software engineer',
                    'Junior web developer',
                    'Project manager with technical background']
}
df_text = pd.DataFrame(text_data)

# String methods
df_text['name_clean'] = df_text['name'].str.title()
df_text['email_domain'] = df_text['email'].str.split('@').str[1]
df_text['name_length'] = df_text['name'].str.len()
df_text['has_experience'] = df_text['description'].str.contains('experience',
    case=False)
df_text['words_count'] = df_text['description'].str.split().str.len()

print("Text processing results:")
print(df_text)
```

Listing 4.5: String manipulation with pandas

4.4 Exercises

1. Create a DataFrame with various types of missing data and practice different handling techniques.
2. Convert a column of string dates to datetime objects and extract year, month, and day.
3. Clean a column of messy string data (mixed case, extra spaces) and standardize the format.
4. Handle a column with mixed numeric and string values, converting everything to numeric.

5. Create a function to automatically detect and report data quality issues in a DataFrame.

4.5 Solutions

1.
```python
# Create DataFrame with missing data
missing_data = {
    'col1': [1, np.nan, 3, None, 5],
    'col2': ['a', 'b', None, 'd', np.nan],
    'col3': [1.1, 2.2, np.nan, np.nan, 5.5]
}
df_missing = pd.DataFrame(missing_data)

# Different handling methods
df_dropped = df_missing.dropna()
df_filled = df_missing.fillna({'col1': 0, 'col2': 'missing', 'col3':
    df_missing['col3'].mean()})
df_ffilled = df_missing.ffill()
```

```python
dates = pd.Series(['2024-01-15', '2024-02-20', '2024-03-25', '2024-04-30'])
date_series = pd.to_datetime(dates)
result = pd.DataFrame({
    'original': dates,
    'datetime': date_series,
    'year': date_series.dt.year,
    'month': date_series.dt.month,
    'day': date_series.dt.day
})
print(result)
```

3.
```python
messy_data = pd.Series(['  JOHN doe  ', 'jane SMITH ', '  bob  ', 'ALICE',
    '  Charlie Wilson  '])
cleaned = messy_data.str.strip().str.title()
print(cleaned)
```

```python
mixed_data = pd.Series([1, '2', 3.0, '4', 'five', 6])
numeric_data = pd.to_numeric(mixed_data, errors='coerce')
print(numeric_data)
```

4.
```python
def data_quality_report(df):
    report = {
        'missing_values': df.isnull().sum(),
        'missing_percentage': (df.isnull().sum() / len(df)) * 100,
        'data_types': df.dtypes,
        'unique_values': df.nunique(),
        'memory_usage': df.memory_usage(deep=True)
    }
    return pd.DataFrame(report)

quality_report = data_quality_report(df_missing)
print(quality_report)
```

Part II

Intermediate Pandas: Data Manipulation

Chapter 5

Data Input/Output Operations

5.1 Reading Different File Formats

Pandas provides excellent support for reading data from various file formats. This section covers the most common ones.

```
import pandas as pd

# Reading CSV files (most common)
df_csv = pd.read_csv('data.csv')
df_csv_custom = pd.read_csv('data.csv', sep=',', header=0, index_col=0)

# Reading Excel files
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')
df_excel_multiple = pd.read_excel('data.xlsx', sheet_name=['Sheet1', 'Sheet2'])

# Reading JSON files
df_json = pd.read_json('data.json')
df_json_lines = pd.read_json('data.jsonl', lines=True)

# Reading from SQL database
import sqlite3
conn = sqlite3.connect('database.db')
df_sql = pd.read_sql_query('SELECT * FROM table_name', conn)

# Reading from URL
df_url = pd.read_csv('https://example.com/data.csv')

# Reading with specific encoding
df_encoded = pd.read_csv('data.csv', encoding='utf-8')
```

Listing 5.1: Reading various file formats with pandas

5.2 Writing Data to Files

```
# Writing to CSV
df.to_csv('output.csv', index=False)

# Writing to Excel
df.to_excel('output.xlsx', sheet_name='Data', index=False)

# Writing to JSON
df.to_json('output.json', orient='records')

# Writing to SQL
from sqlalchemy import create_engine
engine = create_engine('sqlite:///database.db')
df.to_sql('table_name', engine, if_exists='replace', index=False)

```

```
# Writing to Parquet (efficient for large datasets)
df.to_parquet('output.parquet')

# Writing with compression
df.to_csv('output.csv.gz', compression='gzip')
```

Listing 5.2: Writing DataFrames to various file formats

5.3 File Reading Parameters and Options

```
# CSV reading with various options
df = pd.read_csv('data.csv',
                 sep=',',            # Separator
                 header=0,           # Row to use as column names
                 index_col=0,        # Column to use as index
                 usecols=[0, 1, 2], # Columns to read
                 nrows=1000,         # Number of rows to read
                 skiprows=[0, 2],    # Rows to skip
                 na_values=['NA', 'null', ''], # Values to treat as NaN
                 dtype={'column1': str, 'column2': int}, # Data types
                 parse_dates=['date_column'], # Parse dates
                 encoding='utf-8'    # File encoding
                )

# Excel reading options
df = pd.read_excel('data.xlsx',
                   sheet_name=0,         # Sheet name or index
                   header=0,             # Row for column names
                   index_col=0,          # Column for index
                   usecols='A:C,E',      # Columns to read
                   na_values[''],        # Values to treat as NaN
                   dtype={'col1': str}   # Column data types
                  )
```

Listing 5.3: Common parameters for file reading

5.4 Handling Large Files

```
# Reading in chunks
chunk_size = 10000
chunks = []
for chunk in pd.read_csv('large_file.csv', chunksize=chunk_size):
    # Process each chunk
    processed_chunk = chunk[chunk['important_column'] > threshold]
    chunks.append(processed_chunk)

# Combine processed chunks
result = pd.concat(chunks, ignore_index=True)

# Using dtypes for memory optimization
dtype_dict = {
    'id': 'int32',
    'category': 'category',
    'value': 'float32',
    'description': 'string'
}
```

```python
df_optimized = pd.read_csv('data.csv', dtype=dtype_dict)

# Specifying columns to read
columns_to_use = ['id', 'name', 'value']
df_subset = pd.read_csv('data.csv', usecols=columns_to_use)
```

Listing 5.4: Strategies for large datasets

5.5 Exercises

1. Download a sample CSV file from a public URL and read it into pandas.
2. Read an Excel file with multiple sheets and combine them into one DataFrame.
3. Write a DataFrame to both CSV and JSON formats with different formatting options.
4. Create a function that reads a large CSV file in chunks and performs basic analysis on each chunk.
5. Compare the file sizes and read times for CSV, Parquet, and compressed CSV formats.

5.6 Solutions

1.
```python
import pandas as pd

# Read from URL
url = 'https://raw.githubusercontent.com/datasets/iris/master/data/iris.csv'
df_iris = pd.read_csv(url)
print(f"Loaded {len(df_iris)} rows")
print(df_iris.head())
```

```python
# Read multiple sheets and combine
excel_file = 'data.xlsx'
sheets_dict = pd.read_excel(excel_file, sheet_name=None)

# Combine all sheets
all_sheets = []
for sheet_name, sheet_data in sheets_dict.items():
    sheet_data['sheet_name'] = sheet_name
    all_sheets.append(sheet_data)

combined_df = pd.concat(all_sheets, ignore_index=True)
print(combined_df)
```

3.
```python
# Sample DataFrame
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30], 'City': ['NY', 'LA']}
df = pd.DataFrame(data)

# Write to CSV with custom options
df.to_csv('output.csv', index=False, sep='|')

# Write to JSON with different orientations
df.to_json('output_records.json', orient='records', indent=2)
df.to_json('output_values.json', orient='values')
```

```python
def analyze_large_file(file_path, chunk_size=1000):
    chunk_stats = []

    for i, chunk in enumerate(pd.read_csv(file_path, chunksize=chunk_size)):
        stats = {
            'chunk': i + 1,
            'rows': len(chunk),
            'columns': len(chunk.columns),
            'memory_mb': chunk.memory_usage(deep=True).sum() / 1024**2
        }
        chunk_stats.append(stats)
        print(f"Processed chunk {i+1}: {len(chunk)} rows")

    return pd.DataFrame(chunk_stats)

# Usage
stats_df = analyze_large_file('large_data.csv')
print(stats_df)
```

5.

```python
import time
import os

# Create sample data
large_df = pd.DataFrame(np.random.randn(100000, 10),
                        columns=[f'col_{i}' for i in range(10)])

# Test different formats
formats = ['csv', 'parquet', 'csv.gz']
results = []

for fmt in formats:
    filename = f'test_data.{fmt}'

    # Write
    start_write = time.time()
    if fmt == 'csv':
        large_df.to_csv(filename, index=False)
    elif fmt == 'parquet':
        large_df.to_parquet(filename)
    elif fmt == 'csv.gz':
        large_df.to_csv(filename, index=False, compression='gzip')
    write_time = time.time() - start_write

    # Read
    start_read = time.time()
    if fmt == 'csv':
        pd.read_csv(filename)
    elif fmt == 'parquet':
        pd.read_parquet(filename)
    elif fmt == 'csv.gz':
        pd.read_csv(filename, compression='gzip')
    read_time = time.time() - start_read

    # File size
    file_size = os.path.getsize(filename) / 1024  # KB

    results.append({
        'format': fmt,
```

```
        'file_size_kb': file_size,
        'write_time': write_time,
        'read_time': read_time
    })

    # Clean up
    os.remove(filename)

results_df = pd.DataFrame(results)
print(results_df)
```

Chapter 6

Data Aggregation and Group Operations

6.1 GroupBy Fundamentals

```
# Sample sales data
np.random.seed(42)
sales_data = {
    'Region': np.random.choice(['North', 'South', 'East', 'West'], 100),
    'Product': np.random.choice(['Widget A', 'Widget B', 'Gadget C'], 100),
    'Salesperson': np.random.choice(['Alice', 'Bob', 'Charlie', 'Diana'], 100),
    'Sales': np.random.randint(100, 5000, 100),
    'Quantity': np.random.randint(1, 50, 100),
    'Date': pd.date_range('2024-01-01', periods=100, freq='D')
}
sales_df = pd.DataFrame(sales_data)

# Basic groupby
print("Total sales by region:")
region_sales = sales_df.groupby('Region')['Sales'].sum()
print(region_sales)

print("\nAverage sales by product:")
product_avg = sales_df.groupby('Product')['Sales'].mean()
print(product_avg)

# Multiple columns grouping
print("\nSales by region and product:")
region_product = sales_df.groupby(['Region', 'Product'])['Sales'].sum()
print(region_product)
```

Listing 6.1: Basic GroupBy operations

6.2 Aggregation Methods

```
# Multiple aggregation functions
agg_results = sales_df.groupby('Region').agg({
    'Sales': ['sum', 'mean', 'std', 'min', 'max', 'count'],
    'Quantity': ['sum', 'mean']
})
print("Multiple aggregations:")
print(agg_results)

# Custom aggregation functions
def sales_range(x):
    return x.max() - x.min()

def top_sales(x):
```

```
    return x.nlargest(3).sum()

custom_agg = sales_df.groupby('Region').agg({
    'Sales': [sales_range, top_sales, 'mean'],
    'Quantity': ['sum', lambda x: x.mean()]
})
print("\nCustom aggregations:")
print(custom_agg)

# Named aggregations (cleaner syntax)
named_agg = sales_df.groupby('Region').agg(
    total_sales=('Sales', 'sum'),
    avg_sales=('Sales', 'mean'),
    max_quantity=('Quantity', 'max'),
    unique_products=('Product', 'nunique')
)
print("\nNamed aggregations:")
print(named_agg)
```

Listing 6.2: Advanced aggregation techniques

6.3 Transform and Filter Operations

```
# Transform - apply function to each group and return same shape
sales_df['region_avg_sales'] = sales_df.groupby('Region')['Sales'].transform('
    mean')
sales_df['sales_vs_region_avg'] = sales_df['Sales'] - sales_df['region_avg_sales
    ']

print("Data with transformed columns:")
print(sales_df[['Region', 'Sales', 'region_avg_sales', 'sales_vs_region_avg']].
    head())

# Filter - filter groups based on conditions
def high_sales_group(group):
    return group['Sales'].sum() > 50000

high_sales_regions = sales_df.groupby('Region').filter(high_sales_group)
print(f"\nRegions with total sales > 50,000: {high_sales_regions['Region'].
    unique()}")

# Size and count
group_sizes = sales_df.groupby(['Region', 'Product']).size()
print("\nGroup sizes:")
print(group_sizes)
```

Listing 6.3: Transform and filter in GroupBy

6.4 Pivot Tables

```
# Basic pivot table
pivot_simple = pd.pivot_table(sales_df,
                              values='Sales',
                              index='Region',
                              columns='Product',
                              aggfunc='sum')
```

```
print("Simple pivot table:")
print(pivot_simple)

# Multi-level pivot table
pivot_complex = pd.pivot_table(sales_df,
                               values=['Sales', 'Quantity'],
                               index=['Region', 'Salesperson'],
                               columns=['Product'],
                               aggfunc={'Sales': ['sum', 'mean'], 'Quantity': '
                                   sum'},
                               fill_value=0,
                               margins=True)
print("\nComplex pivot table:")
print(pivot_complex)

# Using pivot (vs pivot_table)
pivot_simple_alt = sales_df.pivot_table(index='Region',
                                        columns='Product',
                                        values='Sales',
                                        aggfunc='mean')
print("\nAlternative pivot syntax:")
print(pivot_simple_alt)
```

Listing 6.4: Creating pivot tables

6.5 Cross Tabulation

```
# Create cross tabulation
cross_tab = pd.crosstab(sales_df['Region'], sales_df['Product'])
print("Cross tabulation - counts:")
print(cross_tab)

# Cross tab with margins and normalization
cross_tab_margins = pd.crosstab(sales_df['Region'], sales_df['Product'],
                                margins=True, margins_name='Total')
print("\nCross tabulation with margins:")
print(cross_tab_margins)

# Normalized cross tab
cross_tab_norm = pd.crosstab(sales_df['Region'], sales_df['Product'],
                             normalize='index')  # Normalize by row
print("\nNormalized cross tabulation (by row):")
print(cross_tab_norm)

# Multi-dimensional cross tab
cross_tab_multi = pd.crosstab([sales_df['Region'], sales_df['Salesperson']],
                              sales_df['Product'])
print("\nMulti-dimensional cross tab:")
print(cross_tab_multi)
```

Listing 6.5: Cross tabulation for frequency analysis

6.6 Exercises

1. Create a DataFrame with student data (name, subject, grade, year) and calculate average grades by subject and year.

2. Use transform to center each student's grades around their personal average.

3. Create a pivot table showing average grades by subject and year.

4. Use crosstab to show the count of students by grade category (A: 90-100, B: 80-89, etc.) for each subject.

5. Write a function that performs multiple aggregations on grouped data and returns a formatted report.

6.7 Solutions

1.
```python
# Student data
np.random.seed(42)
students = []
for year in [2022, 2023, 2024]:
    for subject in ['Math', 'Science', 'English', 'History']:
        for _ in range(20):
            students.append({
                'name': f'Student_{np.random.randint(1, 101)}',
                'subject': subject,
                'grade': np.random.randint(60, 101),
                'year': year
            })

students_df = pd.DataFrame(students)

# Average grades by subject and year
avg_grades = students_df.groupby(['subject', 'year'])['grade'].mean().round
    (2)
print(avg_grades)
```

```python
# Center grades around student average
students_df['student_avg'] = students_df.groupby('name')['grade'].transform
    ('mean')
students_df['centered_grade'] = students_df['grade'] - students_df['
    student_avg']
print(students_df[['name', 'grade', 'student_avg', 'centered_grade']].head
    (10))
```

3.
```python
# Pivot table
grades_pivot = pd.pivot_table(students_df,
                              values='grade',
                              index='subject',
                              columns='year',
                              aggfunc='mean')
print(grades_pivot)
```

```python
# Create grade categories
def get_grade_category(grade):
    if grade >= 90: return 'A'
    elif grade >= 80: return 'B'
    elif grade >= 70: return 'C'
    elif grade >= 60: return 'D'
    else: return 'F'
```

```
students_df['grade_category'] = students_df['grade'].apply(
    get_grade_category)

# Cross tabulation
grade_crosstab = pd.crosstab(students_df['subject'],
                             students_df['grade_category'])
print(grade_crosstab)
```

5.

```
def student_performance_report(df):
    report = df.groupby(['subject', 'year']).agg({
        'grade': ['count', 'mean', 'std', 'min', 'max'],
        'name': 'nunique'
    }).round(2)

    # Flatten column names
    report.columns = ['_'.join(col).strip() for col in report.columns.
        values]
    report = report.rename(columns={'name_nunique': 'unique_students'})

    return report

performance_report = student_performance_report(students_df)
print(performance_report)
```

Chapter 7

Data Merging, Joining, and Concatenating

7.1 Concatenation Basics

```python
# Create sample DataFrames
df1 = pd.DataFrame({
    'A': ['A0', 'A1', 'A2', 'A3'],
    'B': ['B0', 'B1', 'B2', 'B3'],
    'C': ['C0', 'C1', 'C2', 'C3']
}, index=[0, 1, 2, 3])

df2 = pd.DataFrame({
    'A': ['A4', 'A5', 'A6', 'A7'],
    'B': ['B4', 'B5', 'B6', 'B7'],
    'C': ['C4', 'C5', 'C6', 'C7']
}, index=[4, 5, 6, 7])

df3 = pd.DataFrame({
    'A': ['A8', 'A9', 'A10', 'A11'],
    'B': ['B8', 'B9', 'B10', 'B11'],
    'C': ['C8', 'C9', 'C10', 'C11']
}, index=[8, 9, 10, 11])

# Basic concatenation (vertical)
result_vertical = pd.concat([df1, df2, df3])
print("Vertical concatenation:")
print(result_vertical)

# Horizontal concatenation
df4 = pd.DataFrame({
    'D': ['D0', 'D1', 'D2', 'D3'],
    'E': ['E0', 'E1', 'E2', 'E3']
}, index=[0, 1, 2, 3])

result_horizontal = pd.concat([df1, df4], axis=1)
print("\nHorizontal concatenation:")
print(result_horizontal)

# Concatenation with keys
result_keys = pd.concat([df1, df2, df3], keys=['X', 'Y', 'Z'])
print("\nConcatenation with keys:")
print(result_keys)
```

Listing 7.1: Concatenating DataFrames

7.2 Merging DataFrames

```python
# Create sample DataFrames for merging
employees = pd.DataFrame({
    'employee_id': [1, 2, 3, 4, 5],
    'name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eva'],
    'department_id': [101, 102, 101, 103, 104]
})

departments = pd.DataFrame({
    'department_id': [101, 102, 103, 105],
    'department_name': ['HR', 'Engineering', 'Marketing', 'Sales'],
    'manager': ['Carol', 'Dave', 'Frank', 'Grace']
})

salaries = pd.DataFrame({
    'employee_id': [1, 2, 3, 6],
    'salary': [50000, 60000, 55000, 70000]
})

# Inner join (default)
inner_join = pd.merge(employees, departments, on='department_id')
print("Inner join:")
print(inner_join)

# Left join
left_join = pd.merge(employees, departments, on='department_id', how='left')
print("\nLeft join:")
print(left_join)

# Right join
right_join = pd.merge(employees, departments, on='department_id', how='right')
print("\nRight join:")
print(right_join)

# Outer join
outer_join = pd.merge(employees, departments, on='department_id', how='outer')
print("\nOuter join:")
print(outer_join)

# Multiple DataFrames
multi_merge = pd.merge(employees, departments, on='department_id').merge(
    salaries, on='employee_id')
print("\nMultiple merges:")
print(multi_merge)
```

Listing 7.2: Merging DataFrames with different join types

7.3 Advanced Merging Techniques

```python
# Merging on multiple columns
df1 = pd.DataFrame({
    'key1': ['A', 'B', 'C', 'D'],
    'key2': [1, 2, 3, 4],
    'value1': ['X1', 'X2', 'X3', 'X4']
})

df2 = pd.DataFrame({
    'key1': ['A', 'B', 'C', 'E'],
```

```
    'key2': [1, 2, 3, 5],
    'value2': ['Y1', 'Y2', 'Y3', 'Y4']
})

merge_multiple_keys = pd.merge(df1, df2, on=['key1', 'key2'])
print("Merge on multiple keys:")
print(merge_multiple_keys)

# Merging with different column names
df3 = pd.DataFrame({
    'id': [1, 2, 3, 4],
    'name': ['Alice', 'Bob', 'Charlie', 'Diana']
})

df4 = pd.DataFrame({
    'emp_id': [1, 2, 5, 6],
    'salary': [50000, 60000, 70000, 80000]
})

merge_different_names = pd.merge(df3, df4, left_on='id', right_on='emp_id')
print("\nMerge with different column names:")
print(merge_different_names)

# Indicator for merge source
merge_with_indicator = pd.merge(employees, departments, on='department_id',
                                how='outer', indicator=True)
print("\nMerge with indicator:")
print(merge_with_indicator)

# Suffixes for overlapping columns
df5 = pd.DataFrame({
    'key': [1, 2, 3],
    'value': [100, 200, 300]
})

df6 = pd.DataFrame({
    'key': [1, 2, 4],
    'value': [400, 500, 600]
})

merge_suffixes = pd.merge(df5, df6, on='key', suffixes=('_left', '_right'))
print("\nMerge with suffixes:")
print(merge_suffixes)
```

Listing 7.3: Advanced merging scenarios

7.4 Joining DataFrames

```
# Create DataFrames with meaningful indexes
left_df = pd.DataFrame({
    'A': ['A0', 'A1', 'A2'],
    'B': ['B0', 'B1', 'B2']
}, index=['K0', 'K1', 'K2'])

right_df = pd.DataFrame({
    'C': ['C0', 'C2', 'C3'],
    'D': ['D0', 'D2', 'D3']
}, index=['K0', 'K2', 'K3'])
```

```python

# Left join on index
left_join_index = left_df.join(right_df, how='left')
print("Left join on index:")
print(left_join_index)

# Right join on index
right_join_index = left_df.join(right_df, how='right')
print("\nRight join on index:")
print(right_join_index)

# Inner join on index
inner_join_index = left_df.join(right_df, how='inner')
print("\nInner join on index:")
print(inner_join_index)

# Outer join on index
outer_join_index = left_df.join(right_df, how='outer')
print("\nOuter join on index:")
print(outer_join_index)

# Join with multiple DataFrames
another_df = pd.DataFrame({
    'E': ['E0', 'E1', 'E2']
}, index=['K0', 'K1', 'K3'])

multi_join = left_df.join([right_df, another_df])
print("\nJoin with multiple DataFrames:")
print(multi_join)
```

Listing 7.4: Joining DataFrames on index

7.5 Exercises

1. Create three DataFrames with customer, order, and product data and merge them appropriately.
2. Concatenate multiple monthly sales reports into a single DataFrame.
3. Perform a merge that shows which customers have no orders (anti-join).
4. Create a DataFrame that combines employee information with their department details using different join types.
5. Compare the performance of merge vs join operations on large datasets.

7.6 Solutions

1.
```python
# Sample data
customers = pd.DataFrame({
    'customer_id': [1, 2, 3, 4],
    'name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'city': ['NY', 'LA', 'Chicago', 'Miami']
})

orders = pd.DataFrame({
```

```
    'order_id': [101, 102, 103, 104],
    'customer_id': [1, 2, 1, 5],
    'product_id': [201, 202, 203, 201],
    'amount': [100, 200, 150, 300]
})

products = pd.DataFrame({
    'product_id': [201, 202, 203, 204],
    'product_name': ['Widget', 'Gadget', 'Tool', 'Accessory'],
    'price': [50, 100, 75, 25]
})

# Merge all three
result = pd.merge(customers, orders, on='customer_id', how='inner')
result = pd.merge(result, products, on='product_id', how='left')
print(result)
```

```
# Create monthly reports
jan_sales = pd.DataFrame({
    'product': ['A', 'B', 'C'],
    'jan_sales': [100, 150, 200]
})

feb_sales = pd.DataFrame({
    'product': ['A', 'B', 'D'],
    'feb_sales': [120, 160, 180]
})

mar_sales = pd.DataFrame({
    'product': ['A', 'C', 'D'],
    'mar_sales': [110, 210, 190]
})

# Concatenate vertically
all_sales = pd.concat([jan_sales, feb_sales, mar_sales], ignore_index=True)
print("Vertical concatenation:")
print(all_sales)

# Merge horizontally
quarterly_sales = jan_sales.merge(feb_sales, on='product', how='outer')
quarterly_sales = quarterly_sales.merge(mar_sales, on='product', how='outer
    ')
print("\nHorizontal merge:")
print(quarterly_sales)
```

3.

```
# Anti-join: customers with no orders
customers_with_orders = pd.merge(customers, orders, on='customer_id', how='
    inner')
all_customers = pd.merge(customers, orders, on='customer_id', how='left',
    indicator=True)
customers_no_orders = all_customers[all_customers['_merge'] == 'left_only']
print("Customers with no orders:")
print(customers_no_orders[['customer_id', 'name']])
```

```
employees = pd.DataFrame({
    'emp_id': [1, 2, 3, 4, 5],
    'name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eva'],
    'dept_id': [101, 102, 101, 103, 104]
```

```
})

departments = pd.DataFrame({
    'dept_id': [101, 102, 103, 105],
    'dept_name': ['HR', 'Engineering', 'Marketing', 'Sales']
})

# Different join types
inner_result = pd.merge(employees, departments, on='dept_id', how='inner')
left_result = pd.merge(employees, departments, on='dept_id', how='left')
right_result = pd.merge(employees, departments, on='dept_id', how='right')
outer_result = pd.merge(employees, departments, on='dept_id', how='outer')

print("Inner join result:")
print(inner_result)
print(f"Rows: {len(inner_result)}")
```

4.

```
import time

# Create larger datasets for performance comparison
large_df1 = pd.DataFrame({
    'key': range(10000),
    'value1': np.random.randn(10000)
})

large_df2 = pd.DataFrame({
    'key': range(5000, 15000),
    'value2': np.random.randn(10000)
})

# Time merge operation
start_time = time.time()
merge_result = pd.merge(large_df1, large_df2, on='key', how='inner')
merge_time = time.time() - start_time

# Time join operation (setting index first)
large_df1_indexed = large_df1.set_index('key')
large_df2_indexed = large_df2.set_index('key')

start_time = time.time()
join_result = large_df1_indexed.join(large_df2_indexed, how='inner')
join_time = time.time() - start_time

print(f"Merge time: {merge_time:.4f} seconds")
print(f"Join time: {join_time:.4f} seconds")
print(f"Merge result shape: {merge_result.shape}")
print(f"Join result shape: {join_result.shape}")
```

Part III

Advanced Pandas: Optimization and Real-World Applications

Chapter 8

Advanced Data Manipulation with MultiIndex

8.1 Creating and Understanding MultiIndex

```
# Creating MultiIndex from arrays
arrays = [
    ['Bar', 'Bar', 'Baz', 'Baz', 'Foo', 'Foo', 'Qux', 'Qux'],
    ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']
]
tuples = list(zip(*arrays))
index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
df_multi = pd.DataFrame(np.random.randn(8, 4), index=index, columns=['A', 'B', '
    C', 'D'])

print("MultiIndex DataFrame:")
print(df_multi)
print(f"\nIndex names: {df_multi.index.names}")
print(f"Index levels: {df_multi.index.levels}")

# Creating MultiIndex from product
index_product = pd.MultiIndex.from_product(
    [['Group1', 'Group2'], ['SubgroupA', 'SubgroupB'], ['Item1', 'Item2']],
    names=['Group', 'Subgroup', 'Item']
)
df_product = pd.DataFrame(np.random.randn(8, 3), index=index_product, columns=['
    X', 'Y', 'Z'])
print("\nMultiIndex from product:")
print(df_product)
```

Listing 8.1: Working with MultiIndex DataFrames

8.2 Indexing with MultiIndex

```
# Different ways to access MultiIndex data
print("First level indexing:")
print(df_multi.loc['Bar'])

print("\nMultiple first level indices:")
print(df_multi.loc[['Bar', 'Foo']])

print("\nSpecific multi-level index:")
print(df_multi.loc[('Bar', 'one')])

print("\nSlicing with MultiIndex:")
print(df_multi.loc[('Bar', 'one'):('Baz', 'two')])

# Using xs for cross-section
```

```
print("\nCross-section at first level:")
print(df_multi.xs('Bar', level='first'))

print("\nCross-section at second level:")
print(df_multi.xs('one', level='second'))

# Boolean indexing with MultiIndex
print("\nBoolean indexing:")
print(df_multi[df_multi['A'] > 0])
```

Listing 8.2: Advanced MultiIndex operations

8.3 Reshaping with Stack and Unstack

```
# Sample data for reshaping
data = {
    'Year': [2022, 2022, 2023, 2023, 2022, 2022, 2023, 2023],
    'Quarter': ['Q1', 'Q2', 'Q1', 'Q2', 'Q1', 'Q2', 'Q1', 'Q2'],
    'Region': ['North', 'North', 'North', 'North', 'South', 'South', 'South', '
       South'],
    'Sales': [100, 150, 120, 180, 90, 130, 110, 160],
    'Profit': [20, 30, 25, 35, 18, 26, 22, 32]
}
df_sales = pd.DataFrame(data)

# Pivot to create MultiIndex columns
pivoted = df_sales.pivot_table(
    index=['Year', 'Quarter'],
    columns='Region',
    values=['Sales', 'Profit']
)
print("Pivoted DataFrame:")
print(pivoted)

# Stack operation
stacked = pivoted.stack()
print("\nStacked DataFrame:")
print(stacked)

# Unstack operation
unstacked = stacked.unstack()
print("\nUnstacked DataFrame:")
print(unstacked)

# Unstack specific level
unstack_region = stacked.unstack('Region')
print("\nUnstacked by Region:")
print(unstack_region)
```

Listing 8.3: Stack and unstack operations

8.4 Advanced MultiIndex Operations

```
# Setting and resetting index
df_reset = df_multi.reset_index()
print("After reset_index:")
```

```
print(df_reset)

df_set = df_reset.set_index(['first', 'second'])
print("\nAfter set_index:")
print(df_set)

# Swapping index levels
df_swapped = df_multi.swaplevel('first', 'second')
print("\nAfter swapping levels:")
print(df_swapped)

# Sorting index
df_sorted = df_multi.sort_index(level='first')
print("\nAfter sorting by first level:")
print(df_sorted)

# MultiIndex on columns
df_multi_cols = pd.DataFrame(
    np.random.randn(4, 8),
    columns=pd.MultiIndex.from_product([['Group1', 'Group2'], ['A', 'B', 'C', 'D
        ']])
)
print("\nMultiIndex columns:")
print(df_multi_cols)
```

Listing 8.4: Complex MultiIndex manipulations

8.5 Exercises

1. Create a MultiIndex DataFrame representing sales data across different regions, products, and time periods.
2. Use stack/unstack to reshape the data between wide and long formats.
3. Calculate summary statistics for each level of the MultiIndex.
4. Filter the MultiIndex DataFrame to show only specific combinations of index levels.
5. Compare the performance of operations on MultiIndex vs single-index DataFrames.

8.6 Solutions

1.
```
# Create comprehensive sales data
index = pd.MultiIndex.from_product([
    ['North', 'South', 'East', 'West'],
    ['Electronics', 'Clothing', 'Home'],
    ['Q1', 'Q2', 'Q3', 'Q4']
], names=['Region', 'Category', 'Quarter'])

sales_data = pd.DataFrame({
    'Revenue': np.random.randint(10000, 100000, 48),
    'Units': np.random.randint(100, 1000, 48),
    'Profit': np.random.randint(1000, 20000, 48)
}, index=index)

print(sales_data.head(10))
```

```python
# Reshape using stack/unstack
# Wide to long
long_format = sales_data.stack()
print("Long format (stacked):")
print(long_format.head(10))

# Long to wide
wide_format = long_format.unstack()
print("\nWide format (unstacked):")
print(wide_format.head())
```

3.

```python
# Summary statistics by different levels
print("By Region:")
print(sales_data.groupby(level='Region').mean())

print("\nBy Category:")
print(sales_data.groupby(level='Category').sum())

print("\nBy Region and Category:")
print(sales_data.groupby(level=['Region', 'Category']).agg(['mean', 'sum',
    'std']))
```

```python
# Filter specific combinations
# Using loc with tuples
filtered = sales_data.loc[
    (['North', 'South'], ['Electronics', 'Clothing'], ['Q1', 'Q2'])
]
print("Filtered data:")
print(filtered)

# Using query (if index levels are properly named)
# filtered_query = sales_data.query('Region in ["North", "South"] and
    Quarter in ["Q1", "Q2"]')
```

5.

```python
import time

# Create equivalent single-index DataFrame
sales_data_single = sales_data.reset_index()

# Compare grouping operations
start_time = time.time()
multi_result = sales_data.groupby(level=['Region', 'Category']).mean()
multi_time = time.time() - start_time

start_time = time.time()
single_result = sales_data_single.groupby(['Region', 'Category']).mean()
single_time = time.time() - start_time

print(f"MultiIndex grouping time: {multi_time:.6f} seconds")
print(f"Single-index grouping time: {single_time:.6f} seconds")
print(f"MultiIndex is {single_time/multi_time:.2f}x faster")
```

Chapter 9

Performance Optimization and Vectorization

9.1 Understanding Vectorization

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

# Create large dataset
np.random.seed(42)
large_df = pd.DataFrame({
    'A': np.random.rand(100000),
    'B': np.random.rand(100000),
    'C': np.random.rand(100000)
})

# Method 1: Using apply (slow)
def slow_calculation(row):
    return row['A'] * row['B'] + row['C']

start_time = time.time()
large_df['slow_result'] = large_df.apply(slow_calculation, axis=1)
slow_time = time.time() - start_time

# Method 2: Vectorized operations (fast)
start_time = time.time()
large_df['fast_result'] = large_df['A'] * large_df['B'] + large_df['C']
fast_time = time.time() - start_time

print(f"Apply method: {slow_time:.4f} seconds")
print(f"Vectorized method: {fast_time:.4f} seconds")
print(f"Speedup: {slow_time/fast_time:.2f}x")

# Verify results are the same
print(f"Results match: {np.allclose(large_df['slow_result'], large_df['
    fast_result'])}")
```

Listing 9.1: Vectorized operations vs loops

9.2 Memory Optimization

```python
# Check memory usage
print("Original memory usage:")
print(large_df.memory_usage(deep=True))
print(f"Total: {large_df.memory_usage(deep=True).sum() / 1024**2:.2f} MB")

# Optimize data types
```

```
def optimize_dtypes(df):
    optimized_df = df.copy()

    # Integer columns
    int_cols = optimized_df.select_dtypes(include=['int']).columns
    for col in int_cols:
        c_min = optimized_df[col].min()
        c_max = optimized_df[col].max()

        if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
            optimized_df[col] = optimized_df[col].astype(np.int8)
        elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
            optimized_df[col] = optimized_df[col].astype(np.int16)
        elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
            optimized_df[col] = optimized_df[col].astype(np.int32)
        else:
            optimized_df[col] = optimized_df[col].astype(np.int64)

    # Float columns
    float_cols = optimized_df.select_dtypes(include=['float']).columns
    for col in float_cols:
        optimized_df[col] = optimized_df[col].astype(np.float32)

    # Object columns to category if low cardinality
    obj_cols = optimized_df.select_dtypes(include=['object']).columns
    for col in obj_cols:
        num_unique = optimized_df[col].nunique()
        num_total = len(optimized_df[col])
        if num_unique / num_total < 0.5:  # If less than 50% unique values
            optimized_df[col] = optimized_df[col].astype('category')

    return optimized_df

# Create DataFrame with mixed types for demonstration
mixed_df = pd.DataFrame({
    'small_ints': np.random.randint(1, 100, 10000),
    'large_ints': np.random.randint(1, 1000000, 10000),
    'floats': np.random.randn(10000),
    'categories': np.random.choice(['A', 'B', 'C', 'D'], 10000),
    'strings': np.random.choice(['cat', 'dog', 'bird', 'fish'], 10000)
})

print("\nBefore optimization:")
print(mixed_df.memory_usage(deep=True))
print(f"Total: {mixed_df.memory_usage(deep=True).sum() / 1024**2:.2f} MB")

optimized_df = optimize_dtypes(mixed_df)

print("\nAfter optimization:")
print(optimized_df.memory_usage(deep=True))
print(f"Total: {optimized_df.memory_usage(deep=True).sum() / 1024**2:.2f} MB")
print(f"Memory reduction: {(1 - optimized_df.memory_usage(deep=True).sum() /
    mixed_df.memory_usage(deep=True).sum()) * 100:.1f}%")
```

Listing 9.2: Reducing memory usage

9.3 Efficient Data Processing

```python
# Method chaining for efficiency
# Inefficient: multiple assignments
df_inefficient = large_df.copy()
df_inefficient = df_inefficient[df_inefficient['A'] > 0.5]
df_inefficient['new_col'] = df_inefficient['B'] * 2
df_inefficient = df_inefficient.sort_values('C')

# Efficient: method chaining
df_efficient = (large_df[large_df['A'] > 0.5]
                .assign(new_col=lambda x: x['B'] * 2)
                .sort_values('C'))

print("Results are identical:", df_inefficient.equals(df_efficient))

# Using query for complex filtering
complex_filtered = large_df.query('A > 0.5 and B < 0.3 and C between 0.1 and 0.9
    ')
print(f"Complex filter result shape: {complex_filtered.shape}")

# Using eval for complex expressions
# Note: eval can be faster for very large DataFrames
result_eval = large_df.eval('D = A + B * C')
print("Using eval:")
print(result_eval[['A', 'B', 'C', 'D']].head())
```

Listing 9.3: Efficient data processing techniques

9.4 Working with Large Datasets

```python
# Processing large files in chunks
def process_large_file(file_path, chunk_size=10000):
    """Process a large CSV file in chunks"""
    processed_chunks = []

    for i, chunk in enumerate(pd.read_csv(file_path, chunksize=chunk_size)):
        # Process each chunk
        chunk_processed = (chunk
                           .query('value > 0')  # Filter rows
                           .assign(processed=True)  # Add column
                           .groupby('category')
                           .agg({'value': ['sum', 'mean']})  # Aggregate
                          )
        processed_chunks.append(chunk_processed)

        if i % 10 == 0:  # Progress indicator
            print(f"Processed {i * chunk_size} rows...")

    # Combine results
    final_result = pd.concat(processed_chunks)
    return final_result

# Using Dask for out-of-core computations (pandas-like interface)
try:
    import dask.dataframe as dd

    # Create a large dataset for demonstration
    large_csv_path = 'large_dataset.csv'
```

```
    # Uncomment to create test file:
    # pd.DataFrame(np.random.randn(1000000, 10)).to_csv(large_csv_path, index=
        False)

    # Read with Dask
    ddf = dd.read_csv(large_csv_path)
    print(f"Dask DataFrame: {len(ddf):,} rows")

    # Perform operations (lazy evaluation)
    result_dask = (ddf[ddf['0'] > 0]
                   .groupby('1')
                   .mean()
                   .compute())  # Actually compute the result

    print("Dask result:")
    print(result_dask.head())

except ImportError:
    print("Dask not installed. Install with: pip install dask")

# Using pandas with efficient data types from start
efficient_dtypes = {
    'user_id': 'int32',
    'product_id': 'int32',
    'category': 'category',
    'value': 'float32',
    'timestamp': 'string'
}

# When reading data, specify dtypes
# df_efficient = pd.read_csv('data.csv', dtype=efficient_dtypes)
```

Listing 9.4: Strategies for large datasets

9.5 Exercises

1. Compare the performance of different methods for calculating row-wise operations on a large DataFrame.
2. Optimize the memory usage of a DataFrame with mixed data types.
3. Process a simulated large dataset using chunk processing and compare with loading the entire dataset.
4. Use method chaining to perform multiple operations in a single expression.
5. Implement a function that automatically suggests optimal data types for a given DataFrame.

9.6 Solutions

1.

```
# Performance comparison
def performance_comparison():
    df = pd.DataFrame(np.random.rand(100000, 5), columns=list('ABCDE'))

    # Method 1: Iterrows (slowest)
    start = time.time()
```

```
    result1 = []
    for idx, row in df.iterrows():
        result1.append(row['A'] + row['B'] * row['C'])
    time1 = time.time() - start

    # Method 2: Apply
    start = time.time()
    result2 = df.apply(lambda row: row['A'] + row['B'] * row['C'], axis=1)
    time2 = time.time() - start

    # Method 3: Vectorized
    start = time.time()
    result3 = df['A'] + df['B'] * df['C']
    time3 = time.time() - start

    print(f"Iterrows: {time1:.4f}s")
    print(f"Apply: {time2:.4f}s")
    print(f"Vectorized: {time3:.4f}s")
    print(f"Vectorized is {time1/time3:.1f}x faster than iterrows")

performance_comparison()
```

```
# Memory optimization
def optimize_dataframe(df):
    # Convert objects to category where appropriate
    for col in df.select_dtypes(include=['object']):
        if df[col].nunique() / len(df) < 0.5:
            df[col] = df[col].astype('category')

    # Downcast numeric columns
    for col in df.select_dtypes(include=['integer']):
        df[col] = pd.to_numeric(df[col], downcast='integer')

    for col in df.select_dtypes(include=['float']):
        df[col] = pd.to_numeric(df[col], downcast='float')

    return df

# Test
test_df = pd.DataFrame({
    'ints': range(10000),
    'small_ints': np.random.randint(1, 100, 10000),
    'floats': np.random.randn(10000),
    'categories': np.random.choice(['A', 'B', 'C'], 10000),
    'text': ['hello'] * 10000
})

original_memory = test_df.memory_usage(deep=True).sum()
optimized_df = optimize_dataframe(test_df.copy())
optimized_memory = optimized_df.memory_usage(deep=True).sum()

print(f"Original: {original_memory / 1024:.1f} KB")
print(f"Optimized: {optimized_memory / 1024:.1f} KB")
print(f"Reduction: {(1 - optimized_memory/original_memory)*100:.1f}%")
```

3.

```
# Chunk processing vs full load
def simulate_large_processing():
    # Create a large CSV for testing
```

```
    large_data = pd.DataFrame(np.random.randn(500000, 5))
    large_data.to_csv('test_large.csv', index=False)

    # Method 1: Full load
    start = time.time()
    full_df = pd.read_csv('test_large.csv')
    full_result = full_df[full_df['0'] > 0].mean()
    full_time = time.time() - start

    # Method 2: Chunk processing
    start = time.time()
    chunks = []
    for chunk in pd.read_csv('test_large.csv', chunksize=10000):
        filtered_chunk = chunk[chunk['0'] > 0]
        chunks.append(filtered_chunk)
    chunk_result = pd.concat(chunks, ignore_index=True).mean()
    chunk_time = time.time() - start

    print(f"Full load time: {full_time:.2f}s")
    print(f"Chunk processing time: {chunk_time:.2f}s")
    print(f"Memory usage full: {full_df.memory_usage().sum() / 1024**2:.1f}
        MB")

    # Clean up
    import os
    os.remove('test_large.csv')

    return full_result, chunk_result

full_res, chunk_res = simulate_large_processing()
print("Results are equal:", full_res.equals(chunk_res))
```

```
# Method chaining example
result = (pd.DataFrame({
            'group': np.random.choice(['A', 'B', 'C'], 1000),
            'value1': np.random.randn(1000),
            'value2': np.random.randn(1000)
        })
        .query('value1 > 0')  # Filter positive values
        .assign(  # Create new columns
            total=lambda x: x['value1'] + x['value2'],
            ratio=lambda x: x['value1'] / x['value2']
        )
        .groupby('group')  # Group by category
        .agg({  # Multiple aggregations
            'value1': ['mean', 'std'],
            'value2': ['min', 'max'],
            'total': 'sum',
            'ratio': 'mean'
        })
        .round(3)  # Round results
       )

print("Method chaining result:")
print(result)
```

4.

```
# Data type suggestion function
def suggest_optimal_dtypes(df):
```

```
    suggestions = {}

    for col in df.columns:
        col_data = df[col]
        dtype = str(col_data.dtype)

        if col_data.dtype == 'object':
            unique_ratio = col_data.nunique() / len(col_data)
            if unique_ratio < 0.5:
                suggestions[col] = 'category'
            else:
                suggestions[col] = 'string'

        elif 'int' in dtype:
            c_min, c_max = col_data.min(), col_data.max()
            if c_min >= 0:  # Unsigned
                if c_max < 256:
                    suggestions[col] = 'uint8'
                elif c_max < 65536:
                    suggestions[col] = 'uint16'
                elif c_max < 4294967296:
                    suggestions[col] = 'uint32'
            else:  # Signed
                if c_min > -128 and c_max < 127:
                    suggestions[col] = 'int8'
                elif c_min > -32768 and c_max < 32767:
                    suggestions[col] = 'int16'
                elif c_min > -2147483648 and c_max < 2147483647:
                    suggestions[col] = 'int32'

        elif 'float' in dtype:
            suggestions[col] = 'float32'

    return suggestions

# Test the function
sample_df = pd.DataFrame({
    'small_ints': range(100),
    'large_ints': range(1000000, 1000100),
    'floats': np.random.randn(100),
    'categories': ['A', 'B'] * 50,
    'strings': [f'text_{i}' for i in range(100)]
})

suggestions = suggest_optimal_dtypes(sample_df)
print("Optimal dtype suggestions:")
for col, suggested_dtype in suggestions.items():
    current_dtype = sample_df[col].dtype
    print(f"{col}: {current_dtype} -> {suggested_dtype}")
```

Chapter 10

Time Series Analysis with Pandas

10.1 Working with DateTime Data

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

# Creating datetime ranges
dates = pd.date_range('2024-01-01', periods=100, freq='D')
print("Date range:")
print(dates)

# Creating time series data
ts_data = pd.DataFrame({
    'date': dates,
    'value': np.random.randn(100).cumsum(),
    'volume': np.random.randint(1000, 5000, 100)
})
ts_data = ts_data.set_index('date')
print("\nTime series data:")
print(ts_data.head())

# DateTime properties
ts_data['year'] = ts_data.index.year
ts_data['month'] = ts_data.index.month
ts_data['day'] = ts_data.index.day
ts_data['dayofweek'] = ts_data.index.dayofweek
ts_data['quarter'] = ts_data.index.quarter
ts_data['is_weekend'] = ts_data.index.dayofweek.isin([5, 6])

print("\nData with DateTime properties:")
print(ts_data.head())
```

Listing 10.1: DateTime operations in pandas

10.2 Time-Based Indexing and Resampling

```python
# Time-based selection
print("January 2024 data:")
print(ts_data['2024-01'])

print("\nFirst week data:")
print(ts_data['2024-01-01':'2024-01-07'])

# Resampling to different frequencies
daily_data = ts_data['value']
weekly_resampled = daily_data.resample('W').mean()
monthly_resampled = daily_data.resample('M').agg(['mean', 'std', 'min', 'max'])

print("\nWeekly resampled:")
```

```
print(weekly_resampled.head())

print("\nMonthly resampled with multiple aggregations:")
print(monthly_resampled.head())

# Resampling with different methods
ohlc_data = daily_data.resample('W').ohlc()
print("\nOHLC resampling:")
print(ohlc_data.head())
```

Listing 10.2: Time-based operations

10.3 Rolling Windows and Expanding Operations

```
# Rolling windows
ts_data['rolling_mean_7'] = ts_data['value'].rolling(window=7).mean()
ts_data['rolling_std_7'] = ts_data['value'].rolling(window=7).std()
ts_data['rolling_min_7'] = ts_data['value'].rolling(window=7).min()
ts_data['rolling_max_7'] = ts_data['value'].rolling(window=7).max()

# Expanding windows
ts_data['expanding_mean'] = ts_data['value'].expanding().mean()
ts_data['expanding_sum'] = ts_data['value'].expanding().sum()

# Custom rolling functions
def custom_roll(x):
    return (x - x.mean()) / x.std()

ts_data['z_score_5'] = ts_data['value'].rolling(window=5).apply(custom_roll)

print("Data with rolling and expanding windows:")
print(ts_data[['value', 'rolling_mean_7', 'expanding_mean', 'z_score_5']].head
    (10))
```

Listing 10.3: Window operations

10.4 Time Series Decomposition

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Create a time series with trend and seasonality
np.random.seed(42)
t = np.arange(365)
trend = 0.1 * t
seasonality = 10 * np.sin(2 * np.pi * t / 30)
noise = np.random.normal(0, 2, 365)
ts_values = trend + seasonality + noise

# Create DataFrame
decomp_df = pd.DataFrame({
    'value': ts_values,
    'date': pd.date_range('2024-01-01', periods=365, freq='D')
}).set_index('date')

# Decompose the time series
```

```
decomposition = seasonal_decompose(decomp_df['value'], model='additive', period
    =30)

# Add components to DataFrame
decomp_df['trend'] = decomposition.trend
decomp_df['seasonal'] = decomposition.seasonal
decomp_df['residual'] = decomposition.resid

print("Decomposition results:")
print(decomp_df.head())

# Check decomposition quality
print(f"\nResidual stats - Mean: {decomp_df['residual'].mean():.4f}, Std: {
    decomp_df['residual'].std():.4f}")
```

Listing 10.4: Time series decomposition

10.5 Lag Features and Time Shifts

```
# Create lag features
for lag in [1, 2, 3, 7, 30]:
    ts_data[f'value_lag_{lag}'] = ts_data['value'].shift(lag)

# Create forward-looking features (for forecasting)
ts_data['value_lead_1'] = ts_data['value'].shift(-1)
ts_data['value_lead_7'] = ts_data['value'].shift(-7)

# Rolling statistics with different windows
windows = [3, 7, 14, 30]
for window in windows:
    ts_data[f'rolling_mean_{window}'] = ts_data['value'].rolling(window).mean()
    ts_data[f'rolling_std_{window}'] = ts_data['value'].rolling(window).std()
    ts_data[f'rolling_min_{window}'] = ts_data['value'].rolling(window).min()
    ts_data[f'rolling_max_{window}'] = ts_data['value'].rolling(window).max()

# Percentage changes
ts_data['daily_pct_change'] = ts_data['value'].pct_change()
ts_data['weekly_pct_change'] = ts_data['value'].pct_change(7)

print("Data with lag features and rolling statistics:")
print(ts_data.select_dtypes(include=[np.number]).head().round(3))
```

Listing 10.5: Creating lag features

10.6 Exercises

1. Create a time series dataset with hourly frequency and add datetime properties.
2. Resample daily data to weekly and monthly frequencies with different aggregation methods.
3. Calculate rolling statistics with multiple window sizes and compare the results.
4. Create lag features for a time series and use them to analyze autocorrelation.
5. Decompose a time series into trend, seasonal, and residual components.

10.7 Solutions

1.
```python
# Hourly time series
hourly_dates = pd.date_range('2024-01-01', periods=24*30, freq='H')
hourly_data = pd.DataFrame({
    'timestamp': hourly_dates,
    'temperature': 20 + 10 * np.sin(2 * np.pi * hourly_dates.hour / 24) +
       np.random.normal(0, 2, len(hourly_dates)),
    'humidity': 50 + 20 * np.random.randn(len(hourly_dates))
})
hourly_data = hourly_data.set_index('timestamp')

# Add datetime properties
hourly_data['hour'] = hourly_data.index.hour
hourly_data['day'] = hourly_data.index.day
hourly_data['weekday'] = hourly_data.index.day_name()
hourly_data['is_night'] = (hourly_data.index.hour < 6) | (hourly_data.index
    .hour > 22)

print(hourly_data.head())
```

```python
# Resample hourly to daily, weekly, monthly
daily_resampled = hourly_data.resample('D').agg({
    'temperature': ['mean', 'min', 'max'],
    'humidity': 'mean'
})

weekly_resampled = hourly_data.resample('W').agg({
    'temperature': ['mean', 'std'],
    'humidity': ['mean', 'std']
})

monthly_resampled = hourly_data.resample('M').agg({
    'temperature': ['mean', 'min', 'max', 'std'],
    'humidity': ['mean', 'min', 'max', 'std']
})

print("Daily resampled:")
print(daily_resampled.head())
```

3.
```python
# Multiple rolling windows
windows = [6, 12, 24, 168]  # 6h, 12h, 24h, 1 week
for window in windows:
    hourly_data[f'temp_rolling_mean_{window}'] = hourly_data['temperature'
       ].rolling(window).mean()
    hourly_data[f'temp_rolling_std_{window}'] = hourly_data['temperature'].
       rolling(window).std()

# Compare smoothing effect
comparison = hourly_data[['temperature'] + [f'temp_rolling_mean_{w}' for w
    in windows]].head(200)
print("Rolling means comparison:")
print(comparison.describe())
```

```python
# Create lag features and analyze autocorrelation
lags = [1, 2, 3, 6, 12, 24]  # 1h to 24h lags
for lag in lags:
```

```
    hourly_data[f'temp_lag_{lag}'] = hourly_data['temperature'].shift(lag)

# Calculate autocorrelation
autocorrelations = {}
for lag in lags:
    autocorr = hourly_data['temperature'].autocorr(lag=lag)
    autocorrelations[lag] = autocorr

print("Autocorrelations by lag:")
for lag, corr in autocorrelations.items():
    print(f"Lag {lag}h: {corr:.3f}")
```

4.

```
# Time series decomposition
from statsmodels.tsa.seasonal import seasonal_decompose

# Use hourly temperature data
temp_series = hourly_data['temperature'].iloc[:24*7]  # First week

# Decompose with daily seasonality (24 hours)
decomposition = seasonal_decompose(temp_series, model='additive', period
    =24)

# Create decomposition DataFrame
decomp_result = pd.DataFrame({
    'original': decomposition.observed,
    'trend': decomposition.trend,
    'seasonal': decomposition.seasonal,
    'residual': decomposition.resid
})

print("Decomposition results:")
print(decomp_result.head(48))  # First two days
print(f"\nResidual statistics:")
print(f"Mean: {decomp_result['residual'].mean():.4f}")
print(f"Std: {decomp_result['residual'].std():.4f}")
```

Chapter 11

Working with Text Data

11.1 Basic String Operations

```
# Create sample text data
text_data = pd.DataFrame({
    'name': ['John Doe', 'Jane Smith', 'Bob Johnson', 'Alice Brown', 'Charlie Wilson'],
    'email': ['john.doe@email.com', 'jane.smith@test.org', 'bob@company.com', 'alice.brown@home.net', 'charlie.wilson@work.io'],
    'description': [
        'Senior Python developer with 5 years experience in data analysis',
        'Data scientist specializing in machine learning and AI',
        'Software engineer with backend development skills',
        'Junior web developer with frontend experience',
        'Project manager with technical background and agile methodology'
    ],
    'phone': ['(123) 456-7890', '987-654-3210', '555-123-4567', '444-555-6666', '333-222-1111']
})

print("Original text data:")
print(text_data)

# Basic string operations
text_data['name_upper'] = text_data['name'].str.upper()
text_data['name_lower'] = text_data['name'].str.lower()
text_data['name_title'] = text_data['name'].str.title()
text_data['name_length'] = text_data['name'].str.len()
text_data['word_count'] = text_data['description'].str.split().str.len()

print("\nAfter basic string operations:")
print(text_data[['name', 'name_upper', 'name_title', 'name_length', 'word_count']])
```

Listing 11.1: String manipulation in pandas

11.2 String Methods and Pattern Matching

```
# String splitting
text_data[['first_name', 'last_name']] = text_data['name'].str.split(' ', expand=True)
text_data['email_domain'] = text_data['email'].str.split('@').str[1]

# Pattern matching and extraction
text_data['has_python'] = text_data['description'].str.contains('python', case=False)
text_data['has_experience'] = text_data['description'].str.contains(r'\d+\s+years?', regex=True)
```

```
text_data['experience_years'] = text_data['description'].str.extract(r'(\d+)\s+years?')[0].fillna(0).astype(int)

# String replacement and cleaning
text_data['phone_clean'] = text_data['phone'].str.replace(r'\D', '', regex=True)  # Remove non-digits
text_data['description_clean'] = text_data['description'].str.replace(r'[^\w\s]', '', regex=True)  # Remove punctuation

# String slicing and positioning
text_data['name_initials'] = text_data['first_name'].str[0] + text_data['last_name'].str[0]
text_data['domain_extension'] = text_data['email_domain'].str.split('.').str[-1]

print("After advanced string operations:")
print(text_data[['name', 'first_name', 'last_name', 'email_domain', 'has_python', 'experience_years', 'phone_clean']])
```

Listing 11.2: Advanced string operations

11.3 Text Analysis and NLP Basics

```
import re
from collections import Counter

# Text analysis functions
def clean_text(text):
    """Clean and preprocess text"""
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    text = re.sub(r'\d+', '', text)      # Remove numbers
    return text.strip()

def get_word_frequencies(series):
    """Get word frequencies from a text series"""
    all_words = ' '.join(series.apply(clean_text)).split()
    return Counter(all_words)

# Apply text cleaning
text_data['description_cleaned'] = text_data['description'].apply(clean_text)

# Word frequency analysis
word_freq = get_word_frequencies(text_data['description'])
print("Most common words in descriptions:")
for word, count in word_freq.most_common(10):
    print(f"{word}: {count}")

# TF-IDF like analysis (simplified)
def calculate_tf(description, word):
    """Calculate term frequency for a word in description"""
    words = clean_text(description).split()
    return words.count(word) / len(words) if words else 0

# Analyze specific terms
terms = ['python', 'data', 'developer', 'machine', 'experience']
for term in terms:
    text_data[f'tf_{term}'] = text_data['description'].apply(lambda x: calculate_tf(x, term))
```

```

print("\nTerm frequencies:")
print(text_data[['name'] + [f'tf_{term}' for term in terms]])
```

Listing 11.3: Text analysis with pandas

11.4 Regular Expressions with Pandas

```
# Complex pattern matching with regex
text_data['job_level'] = text_data['description'].str.extract(r'(senior|junior|lead|principal)', flags=re.IGNORECASE)[0].fillna('mid')
text_data['technologies'] = text_data['description'].str.findall(r'(python|machine learning|AI|web|backend|frontend|agile)', flags=re.IGNORECASE)
text_data['tech_count'] = text_data['technologies'].str.len()

# Email validation and parsing
text_data['is_valid_email'] = text_data['email'].str.match(r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$')
text_data['email_username'] = text_data['email'].str.extract(r'^([^@]+)@')[0]

# Phone number formatting
def format_phone(phone):
    """Standardize phone number format"""
    digits = re.sub(r'\D', '', phone)
    if len(digits) == 10:
        return f"({digits[:3]}) {digits[3:6]}-{digits[6:]}"
    return phone

text_data['phone_formatted'] = text_data['phone'].apply(format_phone)

# Extract skills using regex patterns
skill_patterns = {
    'programming': r'\b(python|java|javascript|c\+\+|c#|ruby|go)\b',
    'data_science': r'\b(machine learning|data science|AI|artificial intelligence|deep learning)\b',
    'web': r'\b(web|frontend|backend|fullstack|react|angular|vue)\b',
    'methodology': r'\b(agile|scrum|kanban|waterfall)\b'
}

for skill_category, pattern in skill_patterns.items():
    text_data[f'has_{skill_category}'] = text_data['description'].str.contains(pattern, case=False, regex=True)

print("Regex extraction results:")
print(text_data[['name', 'job_level', 'tech_count', 'is_valid_email', 'phone_formatted'] +
               [f'has_{cat}' for cat in skill_patterns.keys()]])
```

Listing 11.4: Regular expression operations

11.5 Categorical Text Analysis

```
# Convert text to categorical when appropriate
text_data['job_level'] = text_data['job_level'].astype('category')
text_data['domain_extension'] = text_data['domain_extension'].astype('category')

```

```python
# Analyze categorical text data
print("Job level distribution:")
print(text_data['job_level'].value_counts())

print("\nDomain extension distribution:")
print(text_data['domain_extension'].value_counts())

# Text similarity (basic)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Calculate text similarity between descriptions
vectorizer = TfidfVectorizer(stop_words='english', max_features=50)
tfidf_matrix = vectorizer.fit_transform(text_data['description_cleaned'])
similarity_matrix = cosine_similarity(tfidf_matrix)

# Create similarity DataFrame
similarity_df = pd.DataFrame(
    similarity_matrix,
    index=text_data['name'],
    columns=text_data['name']
)

print("\nText similarity matrix:")
print(similarity_df.round(3))

# Find most similar profiles
def find_similar_profiles(name, n=3):
    """Find n most similar profiles to given name"""
    similarities = similarity_df[name].sort_values(ascending=False)
    return similarities.iloc[1:n+1]  # Exclude self

print("\nSimilar profiles to 'John Doe':")
print(find_similar_profiles('John Doe'))
```

Listing 11.5: Categorical analysis of text data

11.6 Exercises

1. Clean and standardize a dataset of messy customer names and addresses.
2. Extract specific information from text using regular expressions (dates, prices, etc.).
3. Perform sentiment analysis on a set of product reviews.
4. Create word clouds and frequency analysis from text data.
5. Build a simple text classification system using pandas string operations.

11.7 Solutions

1.

```python
# Clean customer data
customers = pd.DataFrame({
    'name': ['  john DOE  ', 'jane SMITH ', '  bob j.  ', 'ALICE-MARIE', '
        charlie wilson  '],
    'address': ['123 main st', '456 ELM AVENUE', '789 oak Rd.', '101 PINE
       BLVD', '202 MAPLE STREET'],
```

```
    'phone': ['(123)456-7890', '987.654.3210', '555-123-4567', '444 555
       6666', '3332221111']
})

# Clean names
customers['name_clean'] = (customers['name']
                           .str.strip()
                           .str.title()
                           .str.replace(r'\s+', ' ', regex=True))

# Clean addresses
customers['address_clean'] = (customers['address']
                              .str.strip()
                              .str.title()
                              .str.replace(r'\.', '', regex=True)
                              .str.replace(r'\s+', ' ', regex=True))

# Clean phones
customers['phone_clean'] = (customers['phone']
                            .str.replace(r'\D', '', regex=True)
                            .str.replace(r'(\d{3})(\d{3})(\d{4})', r'(\1)
                               \2-\3', regex=True))

print("Cleaned customer data:")
print(customers)
```

```
# Extract information from text
texts = pd.Series([
    'Buy now for only $29.99! Limited offer until 2024-12-31',
    'Price: $15.50, available from 2024-01-15 to 2024-06-30',
    'Special discount: $99.00 valid until 2024-03-31',
    'Cost: $45.75, promotion ends 2024-02-28'
])

# Extract prices and dates
extracted_data = pd.DataFrame({
    'original': texts,
    'price': texts.str.extract(r'\$(\d+\.?\d*)')[0].astype(float),
    'start_date': texts.str.extract(r'(\d{4}-\d{2}-\d{2})')[0],
    'end_date': texts.str.extract(r'(\d{4}-\d{2}-\d{2})')[1]
})

print("Extracted information:")
print(extracted_data)
```

3.

```
# Simple sentiment analysis
reviews = pd.DataFrame({
    'review': [
        'This product is amazing! I love it so much.',
        'Terrible quality, very disappointed.',
        'It is okay, nothing special.',
        'Excellent product, would definitely recommend!',
        'Poor customer service and bad product.'
    ]
})

# Simple sentiment dictionary
positive_words = ['amazing', 'love', 'excellent', 'great', 'good', '
    recommend', 'awesome']
```

```
negative_words = ['terrible', 'disappointed', 'poor', 'bad', 'awful', 'horrible']

def simple_sentiment(text):
    text_lower = text.lower()
    positive_count = sum(1 for word in positive_words if word in text_lower)
    negative_count = sum(1 for word in negative_words if word in text_lower)

    if positive_count > negative_count:
        return 'positive'
    elif negative_count > positive_count:
        return 'negative'
    else:
        return 'neutral'

reviews['sentiment'] = reviews['review'].apply(simple_sentiment)
reviews['positive_words'] = reviews['review'].str.lower().str.findall('|'.join(positive_words))
reviews['negative_words'] = reviews['review'].str.lower().str.findall('|'.join(negative_words))

print("Sentiment analysis results:")
print(reviews)
```

```
# Word frequency analysis
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Sample text data
documents = [
    'machine learning data science artificial intelligence',
    'python programming data analysis pandas numpy',
    'deep learning neural networks tensorflow pytorch',
    'data visualization matplotlib seaborn plotly',
    'statistics probability hypothesis testing'
]

# Create word frequency
all_text = ' '.join(documents)
words = all_text.split()
word_freq = pd.Series(words).value_counts()

print("Word frequencies:")
print(word_freq)

# Create simple word cloud (text-based)
print("\nWord cloud (text):")
for word, count in word_freq.head(10).items():
    print(f"{word}: {'#' * count}")

# Alternatively, create actual word cloud image
# wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_text)
# plt.figure(figsize=(10, 5))
# plt.imshow(wordcloud, interpolation='bilinear')
# plt.axis('off')
# plt.show()
```

5.

```python
# Simple text classification
articles = pd.DataFrame({
    'title': [
        'Stock Market Reaches All Time High',
        'Football Team Wins Championship Game',
        'New Technology Company IPO Success',
        'Basketball Player Sets New Record',
        'Economic Growth Exceeds Expectations'
    ],
    'category': ['business', 'sports', 'business', 'sports', 'business']
})

# Create feature vectors based on keywords
business_keywords = ['stock', 'market', 'company', 'economic', 'growth', 'ipo']
sports_keywords = ['football', 'basketball', 'championship', 'game', 'player', 'record']

def extract_features(title):
    title_lower = title.lower()
    return {
        'business_score': sum(1 for word in business_keywords if word in title_lower),
        'sports_score': sum(1 for word in sports_keywords if word in title_lower),
        'length': len(title.split())
    }

# Apply feature extraction
features = articles['title'].apply(extract_features)
feature_df = pd.DataFrame(features.tolist())
articles_with_features = pd.concat([articles, feature_df], axis=1)

# Simple classification rule
def classify_article(row):
    if row['business_score'] > row['sports_score']:
        return 'business'
    elif row['sports_score'] > row['business_score']:
        return 'sports'
    else:
        return 'unknown'

articles_with_features['predicted_category'] = articles_with_features.apply(classify_article, axis=1)
articles_with_features['correct'] = articles_with_features['category'] == articles_with_features['predicted_category']

print("Classification results:")
print(articles_with_features)
print(f"\nAccuracy: {articles_with_features['correct'].mean():.2f}")
```

Chapter 12

Advanced Visualization with Pandas

12.1 Basic Plotting with Pandas

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Create sample data for visualization
np.random.seed(42)
dates = pd.date_range('2024-01-01', periods=100, freq='D')
viz_data = pd.DataFrame({
    'date': dates,
    'sales': np.random.randint(1000, 5000, 100).cumsum(),
    'temperature': 20 + 10 * np.sin(2 * np.pi * np.arange(100) / 30) + np.random
        .normal(0, 3, 100),
    'category': np.random.choice(['A', 'B', 'C'], 100),
    'region': np.random.choice(['North', 'South', 'East', 'West'], 100)
})
viz_data = viz_data.set_index('date')

print("Visualization data:")
print(viz_data.head())

# Basic line plot
plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)
viz_data['sales'].plot(title='Sales Over Time', color='blue')
plt.ylabel('Sales')

plt.subplot(2, 2, 2)
viz_data['temperature'].plot(title='Temperature Over Time', color='red')
plt.ylabel('Temperature')

plt.subplot(2, 2, 3)
viz_data['sales'].hist(bins=20, alpha=0.7, color='green')
plt.title('Sales Distribution')
plt.xlabel('Sales')

plt.subplot(2, 2, 4)
viz_data['category'].value_counts().plot(kind='bar', color='orange')
plt.title('Category Distribution')
plt.xlabel('Category')

plt.tight_layout()
plt.show()
```

Listing 12.1: Basic pandas visualization

12.2 Advanced Plot Types

```python
# Create more complex visualizations
plt.figure(figsize=(15, 10))

# 1. Area plot for cumulative data
plt.subplot(2, 3, 1)
category_sales = viz_data.groupby([viz_data.index.month, 'category'])['sales'].sum().unstack()
category_sales.plot(kind='area', alpha=0.7, title='Monthly Sales by Category')
plt.ylabel('Sales')

# 2. Scatter plot with coloring
plt.subplot(2, 3, 2)
for region in viz_data['region'].unique():
    region_data = viz_data[viz_data['region'] == region]
    plt.scatter(region_data['temperature'], region_data['sales'],
                alpha=0.6, label=region, s=50)
plt.xlabel('Temperature')
plt.ylabel('Sales')
plt.title('Sales vs Temperature by Region')
plt.legend()

# 3. Box plot
plt.subplot(2, 3, 3)
viz_data.boxplot(column='sales', by='category', ax=plt.gca())
plt.title('Sales Distribution by Category')
plt.suptitle('')  # Remove automatic title

# 4. Heatmap (correlation)
plt.subplot(2, 3, 4)
numeric_data = viz_data.select_dtypes(include=[np.number])
correlation_matrix = numeric_data.corr()
plt.imshow(correlation_matrix, cmap='coolwarm', aspect='auto')
plt.colorbar()
plt.xticks(range(len(correlation_matrix.columns)), correlation_matrix.columns, rotation=45)
plt.yticks(range(len(correlation_matrix.columns)), correlation_matrix.columns)
plt.title('Correlation Heatmap')

# 5. Pie chart
plt.subplot(2, 3, 5)
region_counts = viz_data['region'].value_counts()
plt.pie(region_counts.values, labels=region_counts.index, autopct='%1.1f%%')
plt.title('Region Distribution')

# 6. Density plot
plt.subplot(2, 3, 6)
viz_data['sales'].plot(kind='density', color='purple')
plt.title('Sales Density Distribution')
plt.xlabel('Sales')

plt.tight_layout()
plt.show()
```

Listing 12.2: Advanced plotting techniques

12.3 Time Series Visualization

```
# Time series specific plots
plt.figure(figsize=(15, 10))

# 1. Time series with rolling average
plt.subplot(2, 2, 1)
viz_data['sales'].plot(alpha=0.5, label='Daily Sales', color='blue')
viz_data['sales'].rolling(7).mean().plot(label='7-Day Moving Average', color='red', linewidth=2)
plt.title('Sales with Moving Average')
plt.legend()

# 2. Seasonal decomposition visualization
from statsmodels.tsa.seasonal import seasonal_decompose

# Create a sample time series with seasonality
ts_with_seasonality = 100 + 5 * np.arange(100) + 10 * np.sin(2 * np.pi * np.arange(100) / 30)
decomposition = seasonal_decompose(ts_with_seasonality, model='additive', period=30)

plt.subplot(2, 2, 2)
decomposition.observed.plot(title='Original Series', color='blue')
plt.subplot(2, 2, 3)
decomposition.trend.plot(title='Trend', color='green')
plt.subplot(2, 2, 4)
decomposition.seasonal.plot(title='Seasonal', color='red')

plt.tight_layout()
plt.show()

# 3. Seasonal plot
plt.figure(figsize=(12, 6))
# Create monthly data for seasonal analysis
monthly_data = viz_data.resample('M').agg({'sales': 'sum', 'temperature': 'mean'})
monthly_data['month'] = monthly_data.index.month

# Pivot for seasonal plot
seasonal_pivot = monthly_data.pivot_table(values='sales', columns=monthly_data.index.year,
                                          index='month', aggfunc='sum')
seasonal_pivot.plot(marker='o', title='Seasonal Sales Pattern by Year')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.legend(title='Year')
plt.show()
```

Listing 12.3: Time series specific visualizations

12.4 Interactive Visualizations

```
try:
    import plotly.express as px
    import plotly.graph_objects as go
    from plotly.subplots import make_subplots
```

```
    # Create interactive line plot
    fig = px.line(viz_data.reset_index(), x='date', y='sales',
                  title='Interactive Sales Over Time',
                  labels={'sales': 'Sales Amount', 'date': 'Date'})
    fig.show()

    # Create interactive scatter plot
    fig2 = px.scatter(viz_data.reset_index(), x='temperature', y='sales',
                      color='region', size='sales',
                      hover_data=['category'],
                      title='Sales vs Temperature by Region')
    fig2.show()

    # Create interactive bar chart
    monthly_summary = viz_data.resample('M').agg({
        'sales': 'sum',
        'temperature': 'mean'
    }).reset_index()

    fig3 = go.Figure()
    fig3.add_trace(go.Bar(x=monthly_summary['date'], y=monthly_summary['sales'],
                          name='Sales', marker_color='blue'))
    fig3.add_trace(go.Scatter(x=monthly_summary['date'], y=monthly_summary['
        temperature'],
                              name='Temperature', yaxis='y2', line=dict(color='
                                  red')))

    fig3.update_layout(
        title='Monthly Sales and Temperature',
        xaxis=dict(title='Date'),
        yaxis=dict(title='Sales', side='left'),
        yaxis2=dict(title='Temperature', side='right', overlaying='y'),
        legend=dict(x=0, y=1)
    )
    fig3.show()

except ImportError:
    print("Plotly not installed. Install with: pip install plotly")

    # Fallback to matplotlib
    fig, ax1 = plt.subplots(figsize=(12, 6))

    ax1.bar(monthly_summary['date'], monthly_summary['sales'], alpha=0.7, color=
        'blue', label='Sales')
    ax1.set_xlabel('Date')
    ax1.set_ylabel('Sales', color='blue')
    ax1.tick_params(axis='y', labelcolor='blue')

    ax2 = ax1.twinx()
    ax2.plot(monthly_summary['date'], monthly_summary['temperature'], color='red
        ', marker='o', label='Temperature')
    ax2.set_ylabel('Temperature', color='red')
    ax2.tick_params(axis='y', labelcolor='red')

    plt.title('Monthly Sales and Temperature')
    plt.show()
```

Listing 12.4: Interactive plotting with Plotly

12.5 Custom Styling and Themes

```
# Apply custom styles
plt.style.use('seaborn-v0_8')  # Use seaborn style

# Create customized plots
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 10))

# Customized line plot
ax1.plot(viz_data.index, viz_data['sales'],
         color='#2E86AB', linewidth=2, marker='o', markersize=4)
ax1.set_title('Sales Trend', fontsize=14, fontweight='bold')
ax1.set_ylabel('Sales', fontsize=12)
ax1.grid(True, alpha=0.3)
ax1.tick_params(axis='x', rotation=45)

# Customized bar plot
category_performance = viz_data.groupby('category')['sales'].mean().sort_values
    ()
colors = ['#A23B72', '#F18F01', '#C73E1D']
category_performance.plot(kind='barh', ax=ax2, color=colors)
ax2.set_title('Average Sales by Category', fontsize=14, fontweight='bold')
ax2.set_xlabel('Average Sales', fontsize=12)

# Customized scatter plot
scatter = ax3.scatter(viz_data['temperature'], viz_data['sales'],
                      c=pd.factorize(viz_data['region'])[0],
                      cmap='viridis', alpha=0.7, s=50)
ax3.set_title('Sales vs Temperature', fontsize=14, fontweight='bold')
ax3.set_xlabel('Temperature', fontsize=12)
ax3.set_ylabel('Sales', fontsize=12)
plt.colorbar(scatter, ax=ax3, label='Region')

# Customized box plot
box_data = [viz_data[viz_data['region'] == region]['sales'] for region in
    viz_data['region'].unique()]
box_plot = ax4.boxplot(box_data, labels=viz_data['region'].unique(),
                       patch_artist=True)
# Color the boxes
colors = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#96CEB4']
for patch, color in zip(box_plot['boxes'], colors):
    patch.set_facecolor(color)
ax4.set_title('Sales Distribution by Region', fontsize=14, fontweight='bold')
ax4.set_ylabel('Sales', fontsize=12)

plt.tight_layout()
plt.show()

# Save the figure
# plt.savefig('custom_visualizations.png', dpi=300, bbox_inches='tight')
```

Listing 12.5: Customizing plot styles

12.6 Exercises

1. Create a comprehensive dashboard with multiple subplots showing different aspects of your data.

2. Build an interactive visualization that allows exploring relationships between variables.

3. Create a time series visualization with multiple moving averages and trend lines.

4. Design a custom color scheme and apply it to all your visualizations.

5. Compare different plot types for the same data and analyze which conveys information most effectively.

12.7 Solutions

1.
```python
# Comprehensive dashboard
def create_dashboard(df):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))

    # 1. Sales trend
    df['sales'].plot(ax=axes[0,0], color='blue', title='Sales Trend')
    axes[0,0].set_ylabel('Sales')

    # 2. Sales by category
    df.groupby('category')['sales'].sum().plot(kind='bar', ax=axes[0,1],
                                               color=['red', 'green', 'blue'],
                                               title='Total Sales by Category')

    # 3. Temperature distribution
    df['temperature'].hist(ax=axes[0,2], bins=20, alpha=0.7, color='orange')
    axes[0,2].set_title('Temperature Distribution')

    # 4. Sales vs Temperature scatter
    for category in df['category'].unique():
        category_data = df[df['category'] == category]
        axes[1,0].scatter(category_data['temperature'], category_data['sales'],
                          label=category, alpha=0.6)
    axes[1,0].set_xlabel('Temperature')
    axes[1,0].set_ylabel('Sales')
    axes[1,0].set_title('Sales vs Temperature')
    axes[1,0].legend()

    # 5. Monthly sales
    monthly_sales = df['sales'].resample('M').sum()
    monthly_sales.plot(ax=axes[1,1], marker='o', color='purple')
    axes[1,1].set_title('Monthly Sales')
    axes[1,1].tick_params(axis='x', rotation=45)

    # 6. Region distribution
    df['region'].value_counts().plot(kind='pie', ax=axes[1,2], autopct='%1.1f%%')
    axes[1,2].set_title('Region Distribution')

    plt.suptitle('Sales Data Dashboard', fontsize=16, fontweight='bold')
    plt.tight_layout()
    return fig

dashboard = create_dashboard(viz_data)
```

```
plt.show()
```

```
# Interactive exploration (simplified)
try:
    import plotly.express as px

    # Create interactive scatter plot
    interactive_df = viz_data.reset_index()
    fig = px.scatter(interactive_df, x='temperature', y='sales',
                     color='region', size='sales',
                     hover_data=['date', 'category'],
                     title='Interactive Sales Exploration',
                     labels={'sales': 'Sales', 'temperature': 'Temperature'
                        })

    # Add trend line
    fig.add_traces(px.scatter(interactive_df, x='temperature', y='sales',
                              trendline="lowess").data[1])

    fig.show()

except ImportError:
    # Fallback static version
    plt.figure(figsize=(10, 6))
    scatter = plt.scatter(viz_data['temperature'], viz_data['sales'],
                          c=pd.factorize(viz_data['region'])[0],
                          s=viz_data['sales']/100, alpha=0.6)
    plt.colorbar(scatter, label='Region')
    plt.xlabel('Temperature')
    plt.ylabel('Sales')
    plt.title('Sales vs Temperature by Region')
    plt.show()
```

3.

```
# Time series with multiple moving averages
plt.figure(figsize=(12, 8))

# Original series
plt.subplot(2, 1, 1)
viz_data['sales'].plot(color='black', alpha=0.3, label='Daily Sales')

# Multiple moving averages
windows = [7, 14, 30]
colors = ['red', 'blue', 'green']
for window, color in zip(windows, colors):
    viz_data['sales'].rolling(window).mean().plot(
        label=f'{window}-Day MA', color=color, linewidth=2)

plt.title('Sales with Multiple Moving Averages')
plt.legend()
plt.ylabel('Sales')

# Residuals from trend
plt.subplot(2, 1, 2)
trend = viz_data['sales'].rolling(30).mean()
residuals = viz_data['sales'] - trend
residuals.plot(color='purple', alpha=0.7)
plt.axhline(y=0, color='red', linestyle='--', alpha=0.5)
plt.title('Residuals from 30-Day Moving Average')
```

```
plt.ylabel('Residuals')

plt.tight_layout()
plt.show()
```

```
# Custom color scheme
custom_colors = {
    'primary': '#2E86AB',
    'secondary': '#A23B72',
    'accent1': '#F18F01',
    'accent2': '#C73E1D',
    'accent3': '#3C91E6'
}

# Apply custom colors
plt.figure(figsize=(12, 8))

# Plot 1: Line plot with custom colors
plt.subplot(2, 2, 1)
viz_data['sales'].plot(color=custom_colors['primary'], linewidth=2)
plt.title('Sales Trend', color=custom_colors['primary'])
plt.ylabel('Sales')

# Plot 2: Bar plot
plt.subplot(2, 2, 2)
category_data = viz_data.groupby('category')['sales'].mean()
category_data.plot(kind='bar', color=[custom_colors['secondary'],
                                      custom_colors['accent1'],
                                      custom_colors['accent2']])
plt.title('Sales by Category')
plt.xticks(rotation=45)

# Plot 3: Scatter plot
plt.subplot(2, 2, 3)
plt.scatter(viz_data['temperature'], viz_data['sales'],
            alpha=0.6, color=custom_colors['accent3'])
plt.xlabel('Temperature')
plt.ylabel('Sales')
plt.title('Sales vs Temperature')

# Plot 4: Distribution
plt.subplot(2, 2, 4)
viz_data['sales'].hist(bins=20, alpha=0.7, color=custom_colors['accent1'])
plt.title('Sales Distribution')
plt.xlabel('Sales')

plt.suptitle('Custom Styled Visualizations', fontsize=16,
             color=custom_colors['primary'], fontweight='bold')
plt.tight_layout()
plt.show()
```

5.

```
# Plot type comparison
comparison_data = viz_data.groupby('region')['sales'].agg(['mean', 'std', '
    count'])

fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Bar plot
```

```
comparison_data['mean'].plot(kind='bar', ax=axes[0,0], color='skyblue')
axes[0,0].set_title('Bar Plot - Mean Sales by Region')
axes[0,0].tick_params(axis='x', rotation=45)

# Pie chart
axes[0,1].pie(comparison_data['mean'], labels=comparison_data.index,
    autopct='%1.1f%%')
axes[0,1].set_title('Pie Chart - Mean Sales Distribution')

# Box plot
region_sales = [viz_data[viz_data['region'] == region]['sales'] for region
    in comparison_data.index]
axes[1,0].boxplot(region_sales, labels=comparison_data.index)
axes[1,0].set_title('Box Plot - Sales Distribution by Region')
axes[1,0].tick_params(axis='x', rotation=45)

# Violin plot (using boxplot as substitute if violin not available)
try:
    axes[1,1].violinplot(region_sales, showmeans=True)
    axes[1,1].set_xticks(range(1, len(comparison_data.index) + 1))
    axes[1,1].set_xticklabels(comparison_data.index)
    axes[1,1].set_title('Violin Plot - Sales Distribution')
except:
    # Fallback to enhanced box plot
    bp = axes[1,1].boxplot(region_sales, labels=comparison_data.index,
        patch_artist=True)
    for patch in bp['boxes']:
        patch.set_facecolor('lightgreen')
    axes[1,1].set_title('Enhanced Box Plot - Sales Distribution')

plt.suptitle('Comparison of Plot Types for Regional Sales Data', fontsize
    =16)
plt.tight_layout()
plt.show()

# Analysis
print("Effectiveness Analysis:")
print("Bar plot: Good for comparing exact values")
print("Pie chart: Good for showing proportions")
print("Box plot: Good for showing distribution statistics")
print("Violin plot: Best for detailed distribution visualization")
```

Chapter 13

Real-World Case Studies and Best Practices

13.1 Complete Data Analysis Workflow

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def complete_analysis_project():
    """Complete data analysis workflow"""
    print("=== COMPLETE DATA ANALYSIS WORKFLOW ===\n")

    # 1. Data Loading and Exploration
    print("1. DATA LOADING AND EXPLORATION")
    # Simulate real-world dataset
    np.random.seed(42)
    n_customers = 10000

    customer_data = pd.DataFrame({
        'customer_id': range(1, n_customers + 1),
        'age': np.random.randint(18, 70, n_customers),
        'income': np.random.normal(50000, 20000, n_customers).clip(20000,
            150000),
        'city': np.random.choice(['New York', 'Los Angeles', 'Chicago', 'Houston
            ', 'Phoenix'],
                                 n_customers, p=[0.2, 0.3, 0.15, 0.2, 0.15]),
        'signup_date': pd.date_range('2020-01-01', periods=n_customers, freq='H'
            )[:n_customers],
        'loyalty_tier': np.random.choice(['Bronze', 'Silver', 'Gold', 'Platinum'
            ],
                                         n_customers, p=[0.4, 0.3, 0.2, 0.1])
    })

    # Add some missing values to simulate real data
    customer_data.loc[customer_data.sample(frac=0.05).index, 'income'] = np.nan
    customer_data.loc[customer_data.sample(frac=0.03).index, 'city'] = None

    print("Dataset shape:", customer_data.shape)
    print("\nFirst 5 rows:")
    print(customer_data.head())
    print("\nData types:")
    print(customer_data.dtypes)
    print("\nMissing values:")
    print(customer_data.isnull().sum())

    # 2. Data Cleaning and Preprocessing
    print("\n2. DATA CLEANING AND PREPROCESSING")

    # Handle missing values
```

```
    customer_data_clean = customer_data.copy()
    customer_data_clean['income'] = customer_data_clean['income'].fillna(
        customer_data_clean.groupby('loyalty_tier')['income'].transform('median')
    )
    customer_data_clean['city'] = customer_data_clean['city'].fillna('Unknown')

    # Feature engineering
    customer_data_clean['signup_year'] = customer_data_clean['signup_date'].dt.year
    customer_data_clean['signup_month'] = customer_data_clean['signup_date'].dt.month
    customer_data_clean['age_group'] = pd.cut(customer_data_clean['age'],
                                              bins=[18, 25, 35, 45, 55, 65, 70],
                                              labels=['18-25', '26-35', '36-45', '46-55', '56-65', '66+'])
    customer_data_clean['income_category'] = pd.qcut(customer_data_clean['income'],
                                                     q=4, labels=['Low', 'Medium', 'High', 'Very High'])

    print("After cleaning and feature engineering:")
    print(customer_data_clean.info())

    # 3. Exploratory Data Analysis
    print("\n3. EXPLORATORY DATA ANALYSIS")

    # Basic statistics
    print("\nNumerical columns statistics:")
    print(customer_data_clean[['age', 'income']].describe())

    print("\nCategorical columns value counts:")
    for col in ['city', 'loyalty_tier', 'age_group', 'income_category']:
        print(f"\n{col}:")
        print(customer_data_clean[col].value_counts())

    # 4. Advanced Analysis
    print("\n4. ADVANCED ANALYSIS")

    # Customer segmentation analysis
    segmentation = customer_data_clean.groupby(['loyalty_tier', 'age_group']).agg({
        'customer_id': 'count',
        'income': ['mean', 'median', 'std'],
        'age': 'mean'
    }).round(2)

    print("\nCustomer Segmentation Analysis:")
    print(segmentation)

    # Time-based analysis
    monthly_signups = customer_data_clean.groupby(
        [customer_data_clean['signup_date'].dt.to_period('M'), 'loyalty_tier']
    )['customer_id'].count().unstack().fillna(0)

    print("\nMonthly Signups by Loyalty Tier:")
    print(monthly_signups.tail())

    # 5. Visualization
    print("\n5. DATA VISUALIZATION")
```

```
96
97      plt.figure(figsize=(15, 10))
98
99      # Subplot 1: Age distribution
100     plt.subplot(2, 3, 1)
101     customer_data_clean['age'].hist(bins=20, alpha=0.7, color='skyblue')
102     plt.title('Age Distribution')
103     plt.xlabel('Age')
104     plt.ylabel('Count')
105
106     # Subplot 2: Income by loyalty tier
107     plt.subplot(2, 3, 2)
108     customer_data_clean.boxplot(column='income', by='loyalty_tier', ax=plt.gca()
            )
109     plt.title('Income Distribution by Loyalty Tier')
110     plt.suptitle('')
111
112     # Subplot 3: City distribution
113     plt.subplot(2, 3, 3)
114     customer_data_clean['city'].value_counts().plot(kind='bar', color='
            lightgreen')
115     plt.title('Customer Distribution by City')
116     plt.xticks(rotation=45)
117
118     # Subplot 4: Monthly signups trend
119     plt.subplot(2, 3, 4)
120     monthly_signups.plot(ax=plt.gca(), marker='o')
121     plt.title('Monthly Signups Trend')
122     plt.xlabel('Month')
123     plt.ylabel('Number of Signups')
124     plt.legend(title='Loyalty Tier')
125
126     # Subplot 5: Age group by loyalty tier
127     plt.subplot(2, 3, 5)
128     cross_tab = pd.crosstab(customer_data_clean['age_group'],
129                             customer_data_clean['loyalty_tier'],
130                             normalize='index')
131     cross_tab.plot(kind='bar', ax=plt.gca())
132     plt.title('Loyalty Tier Distribution by Age Group')
133     plt.xlabel('Age Group')
134     plt.ylabel('Proportion')
135     plt.legend(title='Loyalty Tier')
136     plt.xticks(rotation=45)
137
138     # Subplot 6: Income vs Age scatter
139     plt.subplot(2, 3, 6)
140     for tier in customer_data_clean['loyalty_tier'].unique():
141         tier_data = customer_data_clean[customer_data_clean['loyalty_tier'] ==
                tier]
142         plt.scatter(tier_data['age'], tier_data['income'], alpha=0.6, label=tier
                , s=20)
143     plt.xlabel('Age')
144     plt.ylabel('Income')
145     plt.title('Income vs Age by Loyalty Tier')
146     plt.legend()
147
148     plt.tight_layout()
149     plt.show()
150
151     # 6. Insights and Recommendations
```

```
    print("\n6. INSIGHTS AND RECOMMENDATIONS")

    # Calculate key metrics
    avg_income_by_tier = customer_data_clean.groupby('loyalty_tier')['income'].
        mean()
    customer_count_by_city = customer_data_clean['city'].value_counts()
    monthly_growth = monthly_signups.sum(axis=1).pct_change().mean()

    print(f"\nKey Insights:")
    print(f"- Average income increases with loyalty tier: {avg_income_by_tier.
        to_dict()}")
    print(f"- Top city for customers: {customer_count_by_city.index[0]} ({
        customer_count_by_city.iloc[0]} customers)")
    print(f"- Average monthly growth rate: {monthly_growth:.2%}")

    # Business recommendations
    print(f"\nBusiness Recommendations:")
    print("1. Focus retention efforts on Platinum and Gold tier customers (
        highest value)")
    print("2. Develop targeted marketing for high-income age groups (45-65)")
    print("3. Expand presence in underpenetrated cities")
    print("4. Create loyalty programs to upgrade Bronze/Silver customers")

    return customer_data_clean, segmentation

# Run the complete analysis
final_data, segmentation_results = complete_analysis_project()
```

Listing 13.1: End-to-end data analysis project

13.2 Performance Optimization Case Study

```
def performance_optimization_demo():
    """Demonstrate performance optimization techniques"""
    print("=== PERFORMANCE OPTIMIZATION CASE STUDY ===\n")

    # Create a large dataset for performance testing
    print("Creating large dataset...")
    n_rows = 1000000
    n_cols = 20

    large_data = pd.DataFrame(
        np.random.randn(n_rows, n_cols),
        columns=[f'feature_{i}' for i in range(n_cols)]
    )

    # Add categorical columns
    large_data['category'] = np.random.choice(['A', 'B', 'C', 'D'], n_rows)
    large_data['subcategory'] = np.random.choice(['X', 'Y', 'Z'], n_rows)
    large_data['flag'] = np.random.choice([True, False], n_rows)

    print(f"Dataset size: {n_rows:,} rows x {n_cols} columns")
    print(f"Memory usage: {large_data.memory_usage(deep=True).sum() / 1024**2:.2
        f} MB")

    # 1. Memory Optimization
    print("\n1. MEMORY OPTIMIZATION")

```

```
    def optimize_memory(df):
        """Optimize DataFrame memory usage"""
        optimized_df = df.copy()

        # Downcast numeric columns
        for col in optimized_df.select_dtypes(include=['integer']):
            optimized_df[col] = pd.to_numeric(optimized_df[col], downcast='
                integer')

        for col in optimized_df.select_dtypes(include=['float']):
            optimized_df[col] = pd.to_numeric(optimized_df[col], downcast='float
                ')

        # Convert object columns to category when appropriate
        for col in optimized_df.select_dtypes(include=['object']):
            num_unique = optimized_df[col].nunique()
            num_total = len(optimized_df[col])
            if num_unique / num_total < 0.5:
                optimized_df[col] = optimized_df[col].astype('category')

        return optimized_df

    import time
    start_time = time.time()
    optimized_data = optimize_memory(large_data)
    optimization_time = time.time() - start_time

    original_memory = large_data.memory_usage(deep=True).sum()
    optimized_memory = optimized_data.memory_usage(deep=True).sum()

    print(f"Original memory: {original_memory / 1024**2:.2f} MB")
    print(f"Optimized memory: {optimized_memory / 1024**2:.2f} MB")
    print(f"Memory reduction: {(1 - optimized_memory/original_memory)*100:.1f}%"
        )
    print(f"Optimization time: {optimization_time:.2f} seconds")

    # 2. Operation Performance Comparison
    print("\n2. OPERATION PERFORMANCE COMPARISON")

    # Test different grouping methods
    operations = []

    # Method 1: Basic groupby
    start_time = time.time()
    result1 = large_data.groupby('category').mean()
    time1 = time.time() - start_time
    operations.append(('Basic groupby', time1))

    # Method 2: Optimized groupby
    start_time = time.time()
    result2 = optimized_data.groupby('category').mean()
    time2 = time.time() - start_time
    operations.append(('Optimized groupby', time2))

    # Method 3: Using categorical
    start_time = time.time()
    large_data_cat = large_data.copy()
    large_data_cat['category'] = large_data_cat['category'].astype('category')
    result3 = large_data_cat.groupby('category').mean()
    time3 = time.time() - start_time
```

```
    operations.append(('Categorical groupby', time3))

    # Print performance results
    print("\nGroupBy Performance:")
    for operation, duration in operations:
        print(f"{operation}: {duration:.4f} seconds")

    # 3. Vectorization vs Apply
    print("\n3. VECTORIZATION VS APPLY")

    # Complex calculation: weighted sum of features
    def complex_calc_apply(row):
        return sum(row[f'feature_{i}'] * (i + 1) for i in range(10))

    def complex_calc_vectorized(df):
        return sum(df[f'feature_{i}'] * (i + 1) for i in range(10))

    # Test apply
    start_time = time.time()
    result_apply = large_data.iloc[:10000].apply(complex_calc_apply, axis=1)
    time_apply = time.time() - start_time

    # Test vectorized
    start_time = time.time()
    result_vectorized = complex_calc_vectorized(large_data.iloc[:10000])
    time_vectorized = time.time() - start_time

    print(f"Apply method: {time_apply:.4f} seconds")
    print(f"Vectorized method: {time_vectorized:.4f} seconds")
    print(f"Speedup: {time_apply/time_vectorized:.1f}x")

    # 4. Chunk Processing for Very Large Data
    print("\n4. CHUNK PROCESSING DEMO")

    def process_in_chunks(df, chunk_size=10000):
        """Process DataFrame in chunks"""
        chunks = []
        total_rows = len(df)

        for i in range(0, total_rows, chunk_size):
            chunk = df.iloc[i:i + chunk_size]
            # Simulate complex processing
            processed_chunk = chunk[chunk['flag'] == True].groupby('category').
                mean()
            chunks.append(processed_chunk)

            if (i // chunk_size) % 10 == 0:
                print(f"Processed {min(i + chunk_size, total_rows):,} rows...")

        # Combine results
        final_result = pd.concat(chunks).groupby(level=0).mean()
        return final_result

    start_time = time.time()
    chunk_result = process_in_chunks(large_data)
    chunk_time = time.time() - start_time

    print(f"Chunk processing time: {chunk_time:.2f} seconds")
    print(f"Chunk processing result shape: {chunk_result.shape}")

```

```
    return optimized_data, operations

# Run performance optimization demo
optimized_df, performance_stats = performance_optimization_demo()
```

Listing 13.2: Performance optimization in practice

13.3 Best Practices and Code Quality

```
def demonstrate_best_practices():
    """Show pandas best practices and coding standards"""
    print("=== PANDAS BEST PRACTICES ===\n")

    # 1. Code Organization
    print("1. CODE ORGANIZATION AND READABILITY")

    # Good: Clean, readable code
    def calculate_customer_metrics(customers_df, orders_df):
        """
        Calculate key customer metrics from customer and order data.

        Parameters:
        customers_df (DataFrame): Customer demographic data
        orders_df (DataFrame): Order transaction data

        Returns:
        DataFrame: Customer metrics with lifetime value and frequency
        """
        # Merge customer and order data
        customer_orders = pd.merge(
            customers_df,
            orders_df,
            on='customer_id',
            how='left'
        )

        # Calculate metrics using method chaining
        customer_metrics = (
            customer_orders
            .groupby('customer_id')
            .agg({
                'order_amount': ['sum', 'mean', 'count'],
                'order_date': ['min', 'max']
            })
            .round(2)
        )

        # Flatten column names
        customer_metrics.columns = [
            'total_spent', 'avg_order_value', 'order_count',
            'first_order', 'last_order'
        ]

        # Calculate additional metrics
        customer_metrics['lifetime_days'] = (
            customer_metrics['last_order'] - customer_metrics['first_order']
        ).dt.days

```

```
        return customer_metrics

    # 2. Error Handling and Validation
    print("\n2. ERROR HANDLING AND VALIDATION")

    def safe_data_processing(df, required_columns=None):
        """
        Safely process DataFrame with validation and error handling.
        """
        if required_columns is None:
            required_columns = ['customer_id', 'order_amount']

        # Input validation
        if not isinstance(df, pd.DataFrame):
            raise TypeError("Input must be a pandas DataFrame")

        missing_columns = [col for col in required_columns if col not in df.
            columns]
        if missing_columns:
            raise ValueError(f"Missing required columns: {missing_columns}")

        try:
            # Create a copy to avoid modifying original data
            processed_df = df.copy()

            # Data validation and cleaning
            processed_df = processed_df.dropna(subset=required_columns)

            # Type validation
            if 'order_amount' in processed_df.columns:
                processed_df['order_amount'] = pd.to_numeric(
                    processed_df['order_amount'], errors='coerce'
                )
                # Remove invalid values
                processed_df = processed_df[processed_df['order_amount'] > 0]

            return processed_df

        except Exception as e:
            print(f"Error during processing: {e}")
            return pd.DataFrame()  # Return empty DataFrame on error

    # 3. Efficient Data Processing Patterns
    print("\n3. EFFICIENT DATA PROCESSING PATTERNS")

    def efficient_data_pipeline(raw_data):
        """
        Demonstrate efficient data processing pipeline.
        """
        return (
            raw_data
            # Initial filtering
            .query('order_amount > 0 and customer_id.notna()')
            # Type conversion
            .assign(
                order_date=lambda x: pd.to_datetime(x['order_date']),
                order_amount=lambda x: pd.to_numeric(x['order_amount'])
            )
            # Feature engineering
            .assign(
```

```
                order_year=lambda x: x['order_date'].dt.year,
                order_month=lambda x: x['order_date'].dt.month,
                amount_category=lambda x: pd.cut(
                    x['order_amount'],
                    bins=[0, 50, 200, 500, float('inf')],
                    labels=['Small', 'Medium', 'Large', 'Very Large']
                )
            )
            # Grouping and aggregation
            .groupby(['customer_id', 'order_year'])
            .agg({
                'order_amount': ['sum', 'mean', 'count'],
                'order_date': ['min', 'max']
            })
            # Flatten columns
            .pipe(lambda df: df.set_axis(
                ['total_amount', 'avg_amount', 'order_count',
                 'first_order', 'last_order'],
                axis=1
            ))
            # Final calculations
            .assign(
                lifetime_days=lambda x: (x['last_order'] - x['first_order']).dt.
                    days,
                daily_spend=lambda x: x['total_amount'] / x['lifetime_days'].
                    clip(1)
            )
            .round(2)
        )

    # 4. Memory and Performance Considerations
    print("\n4. MEMORY AND PERFORMANCE CONSIDERATIONS")

    def memory_efficient_processing(df):
        """
        Process data with memory efficiency in mind.
        """
        # Sample data for memory optimization
        result = {}

        # Use appropriate dtypes from start
        dtypes = {
            'customer_id': 'int32',
            'order_amount': 'float32',
            'category': 'category'
        }

        # Select only needed columns
        needed_columns = ['customer_id', 'order_amount', 'order_date', 'category
            ']
        efficient_df = df[needed_columns].copy()

        # Convert to efficient dtypes
        for col, dtype in dtypes.items():
            if col in efficient_df.columns:
                efficient_df[col] = efficient_df[col].astype(dtype)

        # Process in chunks if very large
        chunk_results = []
        chunk_size = 10000
```

```
166
167         for i in range(0, len(efficient_df), chunk_size):
168             chunk = efficient_df.iloc[i:i + chunk_size]
169
170             # Process chunk
171             chunk_result = (
172                 chunk
173                 .groupby('category')
174                 .agg({'order_amount': 'sum'})
175             )
176             chunk_results.append(chunk_result)
177
178         # Combine results
179         final_result = pd.concat(chunk_results).groupby(level=0).sum()
180
181         result['memory_usage'] = efficient_df.memory_usage(deep=True).sum()
182         result['processing_time'] = "Simulated"
183         result['final_result'] = final_result
184
185         return result
186
187     # 5. Testing and Validation
188     print("\n5. TESTING AND VALIDATION")
189
190     def create_test_data():
191         """Create test data for validation"""
192         return pd.DataFrame({
193             'customer_id': [1, 2, 3, 4, 5],
194             'order_amount': [100.0, 200.0, 150.0, 300.0, 250.0],
195             'order_date': pd.date_range('2024-01-01', periods=5),
196             'category': ['A', 'B', 'A', 'C', 'B']
197         })
198
199     def test_data_processing():
200         """Test the data processing function"""
201         test_df = create_test_data()
202
203         try:
204             result = efficient_data_pipeline(test_df)
205
206             # Validation checks
207             assert len(result) > 0, "Result should not be empty"
208             assert 'total_amount' in result.columns, "Should have total_amount
                column"
209             assert result['total_amount'].sum() > 0, "Total amount should be
                positive"
210
211             print(" All tests passed!")
212             return True
213
214         except Exception as e:
215             print(f" Test failed: {e}")
216             return False
217
218     # Run the test
219     test_passed = test_data_processing()
220
221     print(f"\nSummary:")
222     print("- Write clean, documented code")
223     print("- Use method chaining for readability")
```

```
    print("- Implement proper error handling")
    print("- Optimize for memory and performance")
    print("- Always test your code")
    print("- Use appropriate data types")
    print("- Process large datasets in chunks")

    return test_passed

# Demonstrate best practices
best_practices_result = demonstrate_best_practices()
```

Listing 13.3: Pandas best practices

13.4 Final Project: Complete Business Analysis

```
def final_business_analysis_project():
    """
    Complete business analysis project demonstrating all pandas skills.
    """
    print("=== FINAL BUSINESS ANALYSIS PROJECT ===\n")

    # 1. Problem Statement
    print("1. PROBLEM STATEMENT")
    print("Analyze e-commerce data to identify:")
    print("- Customer segmentation and behavior patterns")
    print("- Sales trends and seasonality")
    print("- Product performance and inventory insights")
    print("- Revenue optimization opportunities")

    # 2. Data Generation (Simulating real business data)
    print("\n2. DATA GENERATION AND LOADING")

    np.random.seed(42)
    n_transactions = 50000
    n_customers = 5000
    n_products = 200

    # Generate customer data
    customers = pd.DataFrame({
        'customer_id': range(1, n_customers + 1),
        'join_date': pd.date_range('2020-01-01', periods=n_customers, freq='H')
            [:n_customers],
        'region': np.random.choice(['North', 'South', 'East', 'West', '
            International'],
                                   n_customers, p=[0.25, 0.25, 0.2, 0.2, 0.1]),
        'customer_tier': np.random.choice(['New', 'Regular', 'VIP'],
                                          n_customers, p=[0.6, 0.35, 0.05]),
        'age': np.random.randint(18, 70, n_customers),
        'lifetime_value': np.random.exponential(1000, n_customers).clip(10,
            10000)
    })

    # Generate product catalog
    products = pd.DataFrame({
        'product_id': range(1, n_products + 1),
        'product_name': [f'Product_{i:03d}' for i in range(1, n_products + 1)],
        'category': np.random.choice(['Electronics', 'Clothing', 'Home', 'Books'
            , 'Sports'],
```

```
                                    n_products, p=[0.3, 0.25, 0.2, 0.15, 0.1]),
        'subcategory': np.random.choice(['A', 'B', 'C', 'D'], n_products),
        'cost_price': np.random.uniform(5, 500, n_products),
        'retail_price': lambda x: x['cost_price'] * np.random.uniform(1.2, 3.0,
            n_products)
    })
    products['retail_price'] = products['cost_price'] * np.random.uniform(1.2,
        3.0, n_products)
    products['margin'] = products['retail_price'] - products['cost_price']

    # Generate transaction data
    transactions = pd.DataFrame({
        'transaction_id': range(1, n_transactions + 1),
        'customer_id': np.random.randint(1, n_customers + 1, n_transactions),
        'product_id': np.random.randint(1, n_products + 1, n_transactions),
        'transaction_date': pd.date_range('2023-01-01', periods=n_transactions,
            freq='H')[:n_transactions],
        'quantity': np.random.randint(1, 5, n_transactions),
        'discount_pct': np.random.choice([0, 0.1, 0.15, 0.2, 0.3],
            n_transactions,
                                         p=[0.5, 0.2, 0.15, 0.1, 0.05])
    })

    # Merge data and calculate final amounts
    transactions = transactions.merge(products[['product_id', 'retail_price']],
        on='product_id')
    transactions['final_price'] = transactions['retail_price'] * (1 -
        transactions['discount_pct'])
    transactions['revenue'] = transactions['final_price'] * transactions['
        quantity']

    print(f"Customers: {len(customers):,}")
    print(f"Products: {len(products):,}")
    print(f"Transactions: {len(transactions):,}")

    # 3. Comprehensive Analysis
    print("\n3. COMPREHENSIVE ANALYSIS")

    # 3.1 Customer Analysis
    print("\n3.1 CUSTOMER ANALYSIS")

    customer_behavior = transactions.merge(customers, on='customer_id')

    customer_metrics = (
        customer_behavior
        .groupby('customer_id')
        .agg({
            'revenue': ['sum', 'mean', 'count'],
            'transaction_date': ['min', 'max', 'nunique'],
            'quantity': 'sum',
            'customer_tier': 'first',
            'region': 'first'
        })
        .pipe(lambda df: df.set_axis([
            'total_revenue', 'avg_transaction', 'transaction_count',
            'first_purchase', 'last_purchase', 'active_days',
            'total_quantity', 'tier', 'region'
        ], axis=1))
        .assign(
```

```
            lifetime_days=lambda x: (x['last_purchase'] - x['first_purchase']).dt.days,
            daily_revenue=lambda x: x['total_revenue'] / x['lifetime_days'].clip(1),
            customer_type=lambda x: np.where(
                x['total_revenue'] > x['total_revenue'].quantile(0.8),
                'High Value',
                np.where(x['total_revenue'] > x['total_revenue'].quantile(0.5),
                    'Medium Value', 'Low Value')
            )
        )
    )

    print("Customer Segments:")
    print(customer_metrics['customer_type'].value_counts())
    print(f"\nAverage Revenue by Tier:")
    print(customer_metrics.groupby('tier')['total_revenue'].mean().round(2))

    # 3.2 Product Analysis
    print("\n3.2 PRODUCT ANALYSIS")

    product_performance = (
        transactions
        .merge(products, on='product_id')
        .groupby(['product_id', 'product_name', 'category'])
        .agg({
            'revenue': 'sum',
            'quantity': 'sum',
            'transaction_id': 'count',
            'retail_price': 'mean',
            'discount_pct': 'mean'
        })
        .rename(columns={'transaction_id': 'units_sold'})
        .assign(
            avg_selling_price=lambda x: x['revenue'] / x['quantity'],
            profit_margin=lambda x: (x['avg_selling_price'] -
                                     products.set_index('product_id').loc[x.index.get_level_values('product_id'), 'cost_price'].values) /
                                    x['avg_selling_price']
        )
        .sort_values('revenue', ascending=False)
    )

    print("Top 10 Products by Revenue:")
    print(product_performance.head(10)[['revenue', 'units_sold', 'profit_margin']].round(2))

    # 3.3 Time Series Analysis
    print("\n3.3 TIME SERIES ANALYSIS")

    # Daily revenue trends
    daily_revenue = (
        transactions
        .set_index('transaction_date')
        .resample('D')
        .agg({'revenue': 'sum', 'transaction_id': 'count'})
        .rename(columns={'transaction_id': 'daily_transactions'})
    )

```

```
    # Add rolling averages
    daily_revenue['revenue_7d_ma'] = daily_revenue['revenue'].rolling(7).mean()
    daily_revenue['revenue_30d_ma'] = daily_revenue['revenue'].rolling(30).mean
        ()

    # Seasonal analysis
    daily_revenue['day_of_week'] = daily_revenue.index.dayofweek
    daily_revenue['month'] = daily_revenue.index.month
    daily_revenue['is_weekend'] = daily_revenue['day_of_week'].isin([5, 6])

    print("Weekly Pattern (Average Revenue by Day):")
    weekly_pattern = daily_revenue.groupby('day_of_week')['revenue'].mean()
    print(weekly_pattern.round(2))

    # 4. Advanced Insights
    print("\n4. ADVANCED INSIGHTS AND RECOMMENDATIONS")

    # Customer Lifetime Value prediction (simplified)
    clv_data = customer_metrics.copy()
    clv_data['predicted_6m_value'] = clv_data['daily_revenue'] * 180

    # Product-category analysis
    category_analysis = (
        product_performance
        .groupby('category')
        .agg({
            'revenue': 'sum',
            'units_sold': 'sum',
            'profit_margin': 'mean'
        })
        .sort_values('revenue', ascending=False)
    )

    # Regional performance
    regional_performance = (
        customer_metrics
        .groupby('region')
        .agg({
            'total_revenue': 'sum',
            'customer_id': 'count',
            'daily_revenue': 'mean'
        })
        .rename(columns={'customer_id': 'customer_count'})
        .assign(
            revenue_per_customer=lambda x: x['total_revenue'] / x['
                customer_count']
        )
    )

    print("\nKey Business Insights:")
    print("=" * 50)

    # Insight 1: Top performing categories
    top_category = category_analysis.index[0]
    top_category_revenue = category_analysis.iloc[0]['revenue']
    print(f"1. {top_category} is the highest revenue category (${
        top_category_revenue:,.0f})")

    # Insight 2: Best customer segment
```

```
    best_segment = customer_metrics.groupby('customer_type')['total_revenue'].mean().idxmax()
    best_segment_value = customer_metrics.groupby('customer_type')['total_revenue'].mean().max()
    print(f"2. {best_segment} customers generate ${best_segment_value:,.0f} on average")

    # Insight 3: Regional performance
    best_region = regional_performance['revenue_per_customer'].idxmax()
    best_region_value = regional_performance['revenue_per_customer'].max()
    print(f"3. {best_region} has highest revenue per customer (${best_region_value:,.0f})")

    # Insight 4: Time-based insights
    best_day = weekly_pattern.idxmax()
    days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
    print(f"4. {days[best_day]} is the best performing day of the week")

    print("\nStrategic Recommendations:")
    print("=" * 50)
    print("1. Focus marketing budget on high-performing categories and regions")
    print("2. Develop retention programs for high-value customer segments")
    print("3. Optimize inventory for weekend and peak day demand")
    print("4. Create targeted promotions for underperforming regions")
    print("5. Implement customer loyalty programs to increase lifetime value")

    # 5. Export Results
    print("\n5. RESULTS EXPORT")

    # Create summary report
    summary_report = {
        'total_revenue': transactions['revenue'].sum(),
        'total_customers': customers['customer_id'].nunique(),
        'total_products': products['product_id'].nunique(),
        'avg_transaction_value': transactions['revenue'].mean(),
        'top_category': top_category,
        'best_region': best_region,
        'customer_acquisition_cost': 50,  # Example metric
        'customer_lifetime_value': clv_data['predicted_6m_value'].mean()
    }

    summary_df = pd.DataFrame([summary_report]).T
    summary_df.columns = ['Value']

    print("\nBusiness Summary:")
    print(summary_df)

    return {
        'customers': customers,
        'products': products,
        'transactions': transactions,
        'customer_metrics': customer_metrics,
        'product_performance': product_performance,
        'daily_revenue': daily_revenue,
        'summary': summary_df
    }

# Run the final project
final_results = final_business_analysis_project()
```

```
257
258 print("\n" + "="*60)
259 print("CONGRATULATIONS! You've completed the Pandas Mastery course!")
260 print("="*60)
261 print("\nYou now have comprehensive skills in:")
262 print("  Data manipulation and cleaning")
263 print("  Advanced analysis and aggregation")
264 print("  Time series analysis")
265 print("  Performance optimization")
266 print("  Real-world business applications")
267 print("  Best practices and professional coding")
268 print("\nContinue practicing with real datasets and explore:")
269 print("- Machine learning integration")
270 print("- Big data technologies (Dask, PySpark)")
271 print("- Database integration (SQL, NoSQL)")
272 print("- Web scraping and API data collection")
273 print("- Advanced visualization libraries")
```

Listing 13.4: Final comprehensive project

13.5 Exercises

1. Implement the complete business analysis project with your own simulated data.
2. Identify three additional business metrics that could be valuable and calculate them.
3. Create a comprehensive dashboard visualizing all key insights from the analysis.
4. Optimize the code for better performance and memory usage.
5. Extend the analysis to include predictive modeling for customer behavior.

13.6 Solutions

1.
```
1  # Custom business analysis implementation
2  def custom_business_analysis():
3      # Generate custom business data
4      np.random.seed(123)
5
6      # Your custom data generation here
7      custom_customers = pd.DataFrame({
8          'customer_id': range(1000),
9          'segment': np.random.choice(['A', 'B', 'C'], 1000),
10         # Add more fields as needed
11     })
12
13     # Implement your analysis pipeline
14     results = {}
15
16     # Add your analysis steps
17     results['customer_summary'] = custom_customers.describe()
18
19     return results
20
21 # Run custom analysis
22 custom_results = custom_business_analysis()
```

```python
# Additional business metrics
def calculate_additional_metrics(transactions, customers):
    additional_metrics = {}

    # 1. Customer retention rate
    monthly_customers = transactions.groupby(
        transactions['transaction_date'].dt.to_period('M')
    )['customer_id'].nunique()
    retention_rate = monthly_customers.pct_change().mean()
    additional_metrics['retention_rate'] = retention_rate

    # 2. Average order value by segment
    segment_aov = transactions.merge(customers, on='customer_id').groupby('
        customer_tier')['revenue'].mean()
    additional_metrics['segment_aov'] = segment_aov

    # 3. Product return rate (simulated)
    # In real scenario, you'd have return data
    additional_metrics['estimated_return_rate'] = 0.05  # 5% assumption

    return additional_metrics

additional_metrics = calculate_additional_metrics(
    final_results['transactions'],
    final_results['customers']
)
print("Additional Metrics:", additional_metrics)
```

3.

```python
# Comprehensive dashboard
def create_business_dashboard(results):
    fig, axes = plt.subplots(3, 2, figsize=(15, 12))

    # 1. Revenue trend
    results['daily_revenue']['revenue'].plot(ax=axes[0,0], title='Daily
        Revenue')

    # 2. Customer segments
    results['customer_metrics']['customer_type'].value_counts().plot(
        kind='pie', ax=axes[0,1], title='Customer Segments'
    )

    # 3. Category performance
    results['product_performance'].groupby('category')['revenue'].sum().
        plot(
        kind='bar', ax=axes[1,0], title='Revenue by Category'
    )

    # 4. Regional performance
    regional_data = results['customer_metrics'].groupby('region')['
        total_revenue'].sum()
    regional_data.plot(kind='bar', ax=axes[1,1], title='Revenue by Region')

    # 5. Price distribution
    results['transactions']['final_price'].hist(ax=axes[2,0], bins=30,
        alpha=0.7)
    axes[2,0].set_title('Price Distribution')

    # 6. Customer lifetime value
    results['customer_metrics']['total_revenue'].plot(
```

```
        kind='box', ax=axes[2,1], title='Customer Lifetime Value
            Distribution'
    )

    plt.suptitle('Business Performance Dashboard', fontsize=16, fontweight=
        'bold')
    plt.tight_layout()
    plt.show()

create_business_dashboard(final_results)
```

```
# Performance optimization
def optimize_business_analysis(transactions, customers, products):
    # Use efficient dtypes
    transactions_opt = transactions.astype({
        'customer_id': 'int32',
        'product_id': 'int32',
        'quantity': 'int16',
        'discount_pct': 'float32',
        'revenue': 'float32'
    })

    customers_opt = customers.astype({
        'customer_id': 'int32',
        'region': 'category',
        'customer_tier': 'category',
        'age': 'int16',
        'lifetime_value': 'float32'
    })

    # Process in chunks for large datasets
    def process_chunked_analysis(df, chunk_size=10000):
        chunks = []
        for i in range(0, len(df), chunk_size):
            chunk = df.iloc[i:i+chunk_size]
            # Perform analysis on chunk
            chunk_analysis = chunk.groupby('customer_id')['revenue'].sum()
            chunks.append(chunk_analysis)
        return pd.concat(chunks).groupby(level=0).sum()

    # Use vectorized operations
    customer_metrics_opt = (
        transactions_opt
        .merge(customers_opt, on='customer_id')
        .groupby('customer_id')
        .agg({
            'revenue': 'sum',
            'transaction_date': ['min', 'max'],
            'quantity': 'sum'
        })
    )

    return {
        'memory_saved': f"{(transactions.memory_usage().sum() -
            transactions_opt.memory_usage().sum()) / 1024**2:.1f} MB",
        'customer_metrics': customer_metrics_opt
    }

optimization_results = optimize_business_analysis(
    final_results['transactions'],
```

```
    final_results['customers'],
    final_results['products']
)
print("Optimization Results:", optimization_results['memory_saved'])
```

5.

```
# Predictive modeling extension
def customer_behavior_prediction(customer_metrics):
    from sklearn.ensemble import RandomForestRegressor
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import mean_absolute_error, r2_score

    # Prepare features for prediction
    features = customer_metrics.select_dtypes(include=[np.number]).copy()
    features = features.fillna(features.mean())

    # Create target variable (future revenue - simplified)
    # In reality, you'd need temporal data for proper prediction
    features['target'] = features['total_revenue'] * np.random.uniform(0.8,
        1.2, len(features))

    X = features.drop('target', axis=1)
    y = features['target']

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
       =0.2, random_state=42)

    # Train model
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)

    # Evaluation
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    feature_importance = pd.DataFrame({
        'feature': X.columns,
        'importance': model.feature_importances_
    }).sort_values('importance', ascending=False)

    return {
        'model': model,
        'mae': mae,
        'r2_score': r2,
        'feature_importance': feature_importance,
        'predictions': y_pred
    }

# Note: This is a simplified example. Real predictive modeling would
   require
# proper temporal validation and feature engineering
try:
    prediction_results = customer_behavior_prediction(final_results['
       customer_metrics'])
    print("Prediction Model Performance:")
    print(f"MAE: {prediction_results['mae']:.2f}")
```

```
    print(f"R² Score: {prediction_results['r2_score']:.2f}")
    print("\nTop Features:")
    print(prediction_results['feature_importance'].head())
except Exception as e:
    print(f"Prediction modeling skipped: {e}")
```

Chapter 14

Working with Different Data Sources

14.1 Database Operations with Pandas

```python
import pandas as pd
import sqlite3
from sqlalchemy import create_engine

# SQLite database operations
def sqlite_operations():
    # Create a connection
    conn = sqlite3.connect('example.db')

    # Read data from SQL table
    df = pd.read_sql_query('SELECT * FROM customers', conn)

    # Write DataFrame to SQL table
    df.to_sql('new_table', conn, if_exists='replace', index=False)

    # Close connection
    conn.close()
    return df

# Using SQLAlchemy for more database types
def sqlalchemy_operations():
    # Create engine for different databases
    # SQLite
    engine_sqlite = create_engine('sqlite:///database.db')

    # PostgreSQL
    # engine_pg = create_engine('postgresql://user:password@localhost:5432/
       database')

    # MySQL
    # engine_mysql = create_engine('mysql://user:password@localhost:3306/
       database')

    # Read from database
    df = pd.read_sql_table('table_name', engine_sqlite)

    # Read with custom query
    query = """
    SELECT customer_id, SUM(amount) as total_spent
    FROM transactions
    WHERE date >= '2024-01-01'
    GROUP BY customer_id
    """
    df_custom = pd.read_sql_query(query, engine_sqlite)

    # Write to database
    df.to_sql('output_table', engine_sqlite, if_exists='replace', index=False)

```

```
    return df

# Advanced database operations
def advanced_db_operations():
    engine = create_engine('sqlite:///example.db')

    # Read in chunks for large tables
    chunk_size = 1000
    chunks = []
    for chunk in pd.read_sql_query('SELECT * FROM large_table',
                                   engine, chunksize=chunk_size):
        processed_chunk = chunk[chunk['value'] > 0]
        chunks.append(processed_chunk)

    large_df = pd.concat(chunks, ignore_index=True)

    # Parameterized queries
    params = {'min_amount': 100, 'start_date': '2024-01-01'}
    safe_query = """
    SELECT * FROM transactions
    WHERE amount > %(min_amount)s
    AND date >= %(start_date)s
    """
    df_safe = pd.read_sql_query(safe_query, engine, params=params)

    return large_df, df_safe
```

Listing 14.1: Working with databases

14.2 Web APIs and JSON Data

```
import requests
import json

def api_operations():
    # Basic API request
    response = requests.get('https://api.example.com/data')

    if response.status_code == 200:
        # Convert JSON response to DataFrame
        data = response.json()
        df = pd.DataFrame(data['results'])

        # Handle nested JSON
        normalized_df = pd.json_normalize(data,
                                          record_path='users',
                                          meta=['page_info'])
        return df, normalized_df

    return pd.DataFrame()

# Working with complex JSON structures
def complex_json_operations():
    # Sample complex JSON
    complex_json = {
        "company": "Tech Corp",
        "employees": [
            {
```

```
                "id": 1,
                "name": "John Doe",
                "department": "Engineering",
                "skills": ["Python", "SQL", "AWS"],
                "contact": {
                    "email": "john@tech.com",
                    "phone": "123-456-7890"
                }
            },
            {
                "id": 2,
                "name": "Jane Smith",
                "department": "Marketing",
                "skills": ["SEO", "Content", "Analytics"],
                "contact": {
                    "email": "jane@tech.com",
                    "phone": "987-654-3210"
                }
            }
        ]
    }

    # Normalize nested JSON
    df_employees = pd.json_normalize(complex_json['employees'])

    # Explode list columns
    df_employees_exploded = df_employees.explode('skills')

    # Create separate DataFrames for nested data
    contacts_df = pd.json_normalize(complex_json['employees'],
                                    record_path=None,
                                    meta=['id'],
                                    record_prefix='contact_')

    return df_employees, df_employees_exploded, contacts_df

# Real API example (using free API)
def real_api_example():
    try:
        # Using JSONPlaceholder free API
        response = requests.get('https://jsonplaceholder.typicode.com/users')

        if response.status_code == 200:
            users_data = response.json()

            # Create main users DataFrame
            users_df = pd.json_normalize(users_data)

            # Extract specific nested data
            addresses_df = pd.json_normalize(users_data,
                                             record_path=None,
                                             meta=['id', 'name'],
                                             record_prefix='address.',
                                             sep='_')

            # Company information
            companies_df = pd.json_normalize(users_data,
                                             record_path=None,
                                             meta=['id'],
                                             record_prefix='company_')
```

```
88
89              print("Users DataFrame shape:", users_df.shape)
90              print("Addresses DataFrame shape:", addresses_df.shape)
91
92              return users_df, addresses_df, companies_df
93
94      except Exception as e:
95          print(f"API error: {e}")
96          return pd.DataFrame(), pd.DataFrame(), pd.DataFrame()
```

Listing 14.2: Working with APIs and JSON

14.3 Web Scraping with Pandas

```
1  import requests
2  from bs4 import BeautifulSoup
3
4  def web_scraping_basics():
5      # Scrape HTML tables from web pages
6      url = 'https://example.com/table-page'
7
8      try:
9          response = requests.get(url)
10         response.raise_for_status()
11
12         # Parse HTML
13         soup = BeautifulSoup(response.content, 'html.parser')
14
15         # Find all tables
16         tables = soup.find_all('table')
17
18         # Read tables into pandas
19         dataframes = []
20         for i, table in enumerate(tables):
21             df = pd.read_html(str(table))[0]
22             df['table_source'] = f'table_{i+1}'
23             dataframes.append(df)
24
25         # Combine all tables
26         combined_df = pd.concat(dataframes, ignore_index=True)
27         return combined_df
28
29     except Exception as e:
30         print(f"Web scraping error: {e}")
31         return pd.DataFrame()
32
33 # Reading HTML tables directly
34 def read_html_tables():
35     # Read tables from HTML file or URL
36     try:
37         # From URL
38         # dfs = pd.read_html('https://en.wikipedia.org/wiki/
               List_of_countries_by_GDP')
39
40         # From HTML string or file
41         html_string = """
42         <table>
43             <tr><th>Name</th><th>Age</th><th>City</th></tr>
```

```
            <tr><td>John</td><td>25</td><td>New York</td></tr>
            <tr><td>Jane</td><td>30</td><td>London</td></tr>
        </table>
        """

        dfs = pd.read_html(html_string)
        return dfs[0] if dfs else pd.DataFrame()

    except Exception as e:
        print(f"HTML reading error: {e}")
        return pd.DataFrame()

# Advanced web scraping with data cleaning
def advanced_web_scraping():
    # Simulated scraping with data cleaning
    scraped_data = []

    # Simulate multiple pages
    for page in range(1, 4):
        # In real scenario, this would be actual web requests
        page_data = {
            'product_name': f'Product {page}',
            'price': 100 * page,
            'rating': 4.0 + (page * 0.1),
            'reviews': page * 10,
            'page_number': page
        }
        scraped_data.append(page_data)

    df = pd.DataFrame(scraped_data)

    # Data cleaning operations
    df['price_category'] = pd.cut(df['price'],
                                  bins=[0, 150, 300, float('inf')],
                                  labels=['Budget', 'Mid-range', 'Premium'])

    df['popularity_score'] = df['rating'] * df['reviews']

    return df
```

Listing 14.3: Web scraping and HTML parsing

14.4 Working with Excel and Office Documents

```
def advanced_excel_operations():
    # Reading multiple sheets
    excel_file = 'data.xlsx'

    # Read all sheets
    all_sheets = pd.read_excel(excel_file, sheet_name=None)

    # Read specific sheets
    specific_sheets = pd.read_excel(excel_file, sheet_name=['Sheet1', 'Sheet2'])

    # Read with specific parameters
    df_custom = pd.read_excel(excel_file,
                              sheet_name=0,
                              header=0,
```

```
                              usecols='A:D,F',
                              skiprows=[0, 2],
                              nrows=1000,
                              dtype={'column1': str, 'column2': float})

    # Writing to Excel with formatting options
    with pd.ExcelWriter('output.xlsx', engine='xlsxwriter') as writer:
        df_custom.to_excel(writer, sheet_name='Processed_Data', index=False)

        # Get workbook and worksheet objects
        workbook = writer.book
        worksheet = writer.sheets['Processed_Data']

        # Add formatting (requires xlsxwriter)
        header_format = workbook.add_format({
            'bold': True,
            'text_wrap': True,
            'valign': 'top',
            'fg_color': '#D7E4BC',
            'border': 1
        })

        # Write headers with format
        for col_num, value in enumerate(df_custom.columns.values):
            worksheet.write(0, col_num, value, header_format)

    return all_sheets, df_custom

# Working with multiple Excel files
def multiple_excel_files():
    import glob

    # Find all Excel files in directory
    excel_files = glob.glob('*.xlsx')

    all_data = []
    for file in excel_files:
        # Read each file
        df = pd.read_excel(file)
        df['source_file'] = file
        all_data.append(df)

    # Combine all data
    combined_data = pd.concat(all_data, ignore_index=True)

    # Summary by source file
    summary = combined_data.groupby('source_file').agg({
        'source_file': 'count',
        **{col: 'mean' for col in combined_data.select_dtypes(include=[np.number
            ]).columns}
    }).rename(columns={'source_file': 'row_count'})

    return combined_data, summary
```

Listing 14.4: Advanced Excel operations

14.5 Cloud Storage and Big Data

```
def cloud_storage_operations():
    # Note: These are examples - actual implementation requires cloud SDKs

    # Amazon S3
    def s3_operations():
        # import boto3
        # s3 = boto3.client('s3')

        # Read from S3
        # df = pd.read_csv('s3://bucket-name/path/to/file.csv')

        # Write to S3
        # df.to_csv('s3://bucket-name/path/to/output.csv', index=False)

        print("S3 operations would require boto3 installation")
        return pd.DataFrame()

    # Google Cloud Storage
    def gcs_operations():
        # from google.cloud import storage
        # client = storage.Client()

        # Read from GCS
        # df = pd.read_csv('gs://bucket-name/path/to/file.csv')

        print("GCS operations would require google-cloud-storage installation")
        return pd.DataFrame()

    # Azure Blob Storage
    def azure_operations():
        # from azure.storage.blob import BlobServiceClient
        # Connect and read/write operations

        print("Azure operations would require azure-storage-blob installation")
        return pd.DataFrame()

    return s3_operations(), gcs_operations(), azure_operations()

# Working with large datasets using Dask
def large_dataset_operations():
    try:
        import dask.dataframe as dd

        # Create large dataset for demonstration
        large_data = pd.DataFrame({
            'id': range(1000000),
            'value1': np.random.randn(1000000),
            'value2': np.random.randn(1000000),
            'category': np.random.choice(['A', 'B', 'C', 'D'], 1000000)
        })

        # Save to CSV for Dask demonstration
        large_data.to_csv('large_dataset.csv', index=False)

        # Read with Dask
        ddf = dd.read_csv('large_dataset.csv')

        # Perform operations (lazy evaluation)
        result = (ddf[ddf['value1'] > 0]
                  .groupby('category')
```

```
                .agg({'value1': 'mean', 'value2': 'std'})
                .compute())  # Actually compute the result

        print("Dask processing completed")
        return result

    except ImportError:
        print("Dask not installed. Install with: pip install dask")

        # Fallback to pandas with chunk processing
        chunks = []
        chunk_size = 10000

        for chunk in pd.read_csv('large_dataset.csv', chunksize=chunk_size):
            processed_chunk = chunk[chunk['value1'] > 0]
            chunks.append(processed_chunk)

        result = pd.concat(chunks, ignore_index=True)
        result = result.groupby('category').agg({
            'value1': 'mean',
            'value2': 'std'
        })

        return result
```

Listing 14.5: Working with cloud storage

14.6 Real-time Data and Streaming

```
def streaming_data_simulation():
    # Simulate streaming data
    import time
    from datetime import datetime

    streaming_data = []

    for i in range(100):
        # Simulate real-time data point
        data_point = {
            'timestamp': datetime.now(),
            'value': np.random.randn(),
            'sensor_id': f'sensor_{i % 5}',
            'batch_number': i // 20
        }
        streaming_data.append(data_point)

        # Simulate time delay
        time.sleep(0.01)

    # Convert to DataFrame
    stream_df = pd.DataFrame(streaming_data)

    # Real-time analytics
    latest_values = stream_df.groupby('sensor_id').last()
    rolling_avg = (stream_df.set_index('timestamp')
                   .groupby('sensor_id')['value']
                   .rolling('5s')
                   .mean()
```

```
                    .reset_index())

    return stream_df, latest_values, rolling_avg

# Data pipeline simulation
def data_pipeline_simulation():
    # Simulate complete data pipeline
    print("1. Data Extraction")
    raw_data = pd.DataFrame({
        'user_id': range(1000),
        'action': np.random.choice(['click', 'view', 'purchase'], 1000),
        'timestamp': pd.date_range('2024-01-01', periods=1000, freq='T'),
        'value': np.random.randint(1, 100, 1000)
    })

    print("2. Data Transformation")
    transformed_data = (raw_data
                        .assign(
                            hour=lambda x: x['timestamp'].dt.hour,
                            date=lambda x: x['timestamp'].dt.date,
                            value_category=lambda x: pd.cut(x['value'],
                                                            bins=[0, 25, 50, 75,
                                                                100],
                                                            labels=['Low', 'Medium'
                                                                , 'High', 'Very
                                                                High'])
                        )
                        .query('value > 10')
                       )

    print("3. Data Loading")
    # Simulate loading to database or file
    transformed_data.to_csv('processed_data.csv', index=False)

    print("4. Data Analysis")
    analysis = transformed_data.groupby(['date', 'action']).agg({
        'user_id': 'count',
        'value': ['mean', 'sum']
    }).round(2)

    return raw_data, transformed_data, analysis
```

Listing 14.6: Handling real-time data

14.7 Exercises

1. Connect to a SQLite database, create a table, and perform basic CRUD operations using pandas.

2. Fetch data from a public API and transform the JSON response into a structured DataFrame.

3. Scrape data from a webpage containing tables and clean the extracted data.

4. Create an Excel file with multiple sheets and different formatting options.

5. Build a complete data pipeline that extracts, transforms, and loads data from multiple sources.

14.8 Solutions

1.

```python
def sqlite_exercise():
    import sqlite3

    # Create database and table
    conn = sqlite3.connect('exercise.db')

    # Sample data
    employees = pd.DataFrame({
        'id': range(1, 6),
        'name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eva'],
        'department': ['HR', 'Engineering', 'Engineering', 'Marketing', 'HR'],
        'salary': [50000, 80000, 75000, 60000, 55000],
        'join_date': pd.date_range('2020-01-01', periods=5, freq='M')
    })

    # Write to database
    employees.to_sql('employees', conn, if_exists='replace', index=False)

    # Read with query
    query = "SELECT * FROM employees WHERE salary > 60000"
    high_earners = pd.read_sql_query(query, conn)

    # Update data
    cursor = conn.cursor()
    cursor.execute("UPDATE employees SET salary = 85000 WHERE name = 'Bob'")
    conn.commit()

    # Read updated data
    updated_employees = pd.read_sql_query("SELECT * FROM employees", conn)

    conn.close()
    return employees, high_earners, updated_employees

emp_df, high_earners_df, updated_df = sqlite_exercise()
print("High earners:", high_earners_df)
```

```python
def api_exercise():
    import requests

    try:
        # Use JSONPlaceholder API
        response = requests.get('https://jsonplaceholder.typicode.com/posts')

        if response.status_code == 200:
            posts_data = response.json()

            # Convert to DataFrame
            posts_df = pd.DataFrame(posts_data)

            # Get user data
            users_response = requests.get('https://jsonplaceholder.typicode.com/users')
            users_data = users_response.json()
            users_df = pd.json_normalize(users_data)
```

```
            # Merge posts with user information
            merged_df = posts_df.merge(users_df[['id', 'name', 'email']],
                                       left_on='userId', right_on='id',
                                       suffixes=('_post', '_user'))

            # Analysis
            posts_per_user = merged_df.groupby('name').agg({
                'id_post': 'count',
                'title': lambda x: ', '.join(x[:2])  # First two titles
            }).rename(columns={'id_post': 'post_count', 'title': '
                sample_titles'})

            return merged_df, posts_per_user

    except Exception as e:
        print(f"API exercise error: {e}")
        return pd.DataFrame(), pd.DataFrame()

posts_df, user_summary = api_exercise()
print("Posts per user:", user_summary)
```

3.

```
def web_scraping_exercise():
    # For this exercise, we'll create HTML content and scrape it
    html_content = """
    <html>
        <body>
            <table border="1">
                <tr>
                    <th>Product</th>
                    <th>Price</th>
                    <th>Rating</th>
                    <th>Stock</th>
                </tr>
                <tr>
                    <td>Laptop</td>
                    <td>$999.99</td>
                    <td>4.5</td>
                    <td>15</td>
                </tr>
                <tr>
                    <td>Mouse</td>
                    <td>$29.99</td>
                    <td>4.2</td>
                    <td>50</td>
                </tr>
                <tr>
                    <td>Keyboard</td>
                    <td>$79.99</td>
                    <td>4.7</td>
                    <td>25</td>
                </tr>
            </table>
        </body>
    </html>
    """

    # Read HTML table
    dfs = pd.read_html(html_content)
```

```
    products_df = dfs[0] if dfs else pd.DataFrame()

    # Data cleaning
    if not products_df.empty:
        products_df['Price'] = products_df['Price'].str.replace('$', '').
            astype(float)
        products_df['Rating'] = products_df['Rating'].astype(float)
        products_df['Stock'] = products_df['Stock'].astype(int)

        # Add calculated columns
        products_df['Value_Score'] = (products_df['Rating'] * products_df['
            Price'] / 100).round(2)
        products_df['Stock_Status'] = np.where(products_df['Stock'] > 30, '
            High',
                                               np.where(products_df['Stock']
                                                   > 10, 'Medium', 'Low'))

    return products_df

cleaned_products = web_scraping_exercise()
print("Cleaned products data:")
print(cleaned_products)
```

```
def excel_exercise():
    # Create sample data for different departments
    sales_data = pd.DataFrame({
        'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
        'Revenue': [10000, 12000, 11000, 13000, 14000],
        'Expenses': [8000, 8500, 8200, 9000, 9500],
        'Profit': [2000, 3500, 2800, 4000, 4500]
    })

    hr_data = pd.DataFrame({
        'Employee': ['Alice', 'Bob', 'Charlie', 'Diana'],
        'Department': ['Sales', 'Engineering', 'Engineering', 'Marketing'],
        'Salary': [50000, 80000, 75000, 60000],
        'Join_Date': pd.to_datetime(['2020-01-15', '2019-03-20', '
            2021-07-10', '2020-11-05'])
    })

    inventory_data = pd.DataFrame({
        'Product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor'],
        'Category': ['Electronics', 'Accessories', 'Accessories', '
            Electronics'],
        'Stock': [50, 200, 150, 30],
        'Price': [999.99, 29.99, 79.99, 299.99],
        'Last_Restock': pd.to_datetime(['2024-01-10', '2024-01-15', '
            2024-01-12', '2024-01-08'])
    })

    # Write to Excel with different sheets
    with pd.ExcelWriter('company_data.xlsx', engine='xlsxwriter') as writer
        :
        # Sales data sheet
        sales_data.to_excel(writer, sheet_name='Sales', index=False)

        # HR data sheet
        hr_data.to_excel(writer, sheet_name='HR', index=False)

        # Inventory data sheet
```

```
        inventory_data.to_excel(writer, sheet_name='Inventory', index=False
            )

        # Get workbook
        workbook = writer.book

        # Add formats
        money_format = workbook.add_format({'num_format': '$#,##0.00'})
        date_format = workbook.add_format({'num_format': 'yyyy-mm-dd'})
        header_format = workbook.add_format({
            'bold': True,
            'bg_color': '#366092',
            'font_color': 'white'
        })

        # Apply formats to each sheet
        for sheet_name in ['Sales', 'HR', 'Inventory']:
            worksheet = writer.sheets[sheet_name]

            # Format headers
            for col_num, value in enumerate(writer.sheets[sheet_name].table
                .values[0]):
                worksheet.write(0, col_num, value, header_format)

            # Auto-adjust column widths
            worksheet.set_column('A:Z', 15)

    print("Excel file created: company_data.xlsx")
    return sales_data, hr_data, inventory_data

sales, hr, inventory = excel_exercise()
```

4.

```
def data_pipeline_exercise():
    print("=== COMPLETE DATA PIPELINE EXERCISE ===\n")

    # Step 1: Extract from multiple sources
    print("1. EXTRACTING DATA...")

    # Source 1: CSV file
    try:
        customer_data = pd.DataFrame({
            'customer_id': [1, 2, 3, 4, 5],
            'name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eva'],
            'join_date': pd.date_range('2023-01-01', periods=5, freq='M'),
            'region': ['North', 'South', 'North', 'East', 'West']
        })
    except:
        customer_data = pd.DataFrame()

    # Source 2: Simulated API data
    try:
        orders_data = pd.DataFrame({
            'order_id': [101, 102, 103, 104, 105],
            'customer_id': [1, 2, 1, 3, 4],
            'order_date': pd.date_range('2024-01-01', periods=5, freq='D'),
            'amount': [150.0, 200.0, 75.0, 300.0, 125.0],
            'status': ['completed', 'completed', 'pending', 'completed', '
                shipped']
        })
```

```
    except:
        orders_data = pd.DataFrame()

    # Source 3: Simulated database data
    try:
        products_data = pd.DataFrame({
            'product_id': [201, 202, 203, 204],
            'product_name': ['Laptop', 'Mouse', 'Keyboard', 'Monitor'],
            'category': ['Electronics', 'Accessories', 'Accessories', '
               Electronics'],
            'price': [999.99, 29.99, 79.99, 299.99]
        })
    except:
        products_data = pd.DataFrame()

    print(f"Extracted: {len(customer_data)} customers, {len(orders_data)}
       orders, {len(products_data)} products")

    # Step 2: Transform data
    print("\n2. TRANSFORMING DATA...")

    # Clean and merge data
    if not orders_data.empty and not customer_data.empty:
        # Merge orders with customer data
        orders_enriched = orders_data.merge(customer_data, on='customer_id'
            )

        # Add calculated fields
        orders_enriched['order_weekday'] = orders_enriched['order_date'].dt
            .day_name()
        orders_enriched['amount_category'] = pd.cut(orders_enriched['amount
            '],
                                                    bins=[0, 100, 200, float(
                                                        'inf')],
                                                    labels=['Small', 'Medium'
                                                        , 'Large'])

        # Aggregate data
        customer_summary = orders_enriched.groupby(['customer_id', 'name',
            'region']).agg({
            'order_id': 'count',
            'amount': ['sum', 'mean'],
            'order_date': ['min', 'max']
        }).round(2)

        # Flatten column names
        customer_summary.columns = ['total_orders', 'total_spent', '
            avg_order_value',
                                    'first_order', 'last_order']

        customer_summary = customer_summary.reset_index()

        # Regional analysis
        regional_summary = orders_enriched.groupby('region').agg({
            'order_id': 'count',
            'amount': ['sum', 'mean'],
            'customer_id': 'nunique'
        }).round(2)

        regional_summary.columns = ['total_orders', 'total_revenue', '
```

```
            avg_order_value', 'unique_customers']
        regional_summary = regional_summary.reset_index()

    # Step 3: Load data
    print("\n3. LOADING DATA...")

    # Save to different formats
    if 'orders_enriched' in locals():
        orders_enriched.to_csv('processed_orders.csv', index=False)
        customer_summary.to_csv('customer_analysis.csv', index=False)
        regional_summary.to_csv('regional_analysis.csv', index=False)

        # Save to Excel with multiple sheets
        with pd.ExcelWriter('business_intelligence.xlsx') as writer:
            orders_enriched.to_excel(writer, sheet_name='Orders_Detailed',
                index=False)
            customer_summary.to_excel(writer, sheet_name='Customer_Summary'
                , index=False)
            regional_summary.to_excel(writer, sheet_name='Regional_Analysis
                ', index=False)

    print("Pipeline completed successfully!")

    # Return results
    results = {
        'customer_data': customer_data,
        'orders_data': orders_data,
        'products_data': products_data
    }

    if 'orders_enriched' in locals():
        results.update({
            'orders_enriched': orders_enriched,
            'customer_summary': customer_summary,
            'regional_summary': regional_summary
        })

    return results

# Run the pipeline
pipeline_results = data_pipeline_exercise()
```

Glossary

pandas A Python library for data manipulation and analysis providing data structures and operations for manipulating numerical tables and time series.

DataFrame Two-dimensional labeled data structure with columns of potentially different types, similar to a spreadsheet or SQL table.

Series One-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.).

NaN Not a Number, a special floating-point value used to represent missing or undefined data in pandas.

GroupBy A pandas operation that involves splitting data into groups based on some criteria, applying a function to each group, and combining the results.

Index An immutable array implementing an ordered, sliceable set for labeling DataFrame and Series objects.

Boolean Indexing Using boolean arrays to filter data in pandas objects.

Vectorization Performing operations on entire arrays rather than individual elements, for improved performance.

Method Chaining Technique of calling multiple methods in sequence on the same object.

Reshaping Changing the structure of data from wide to long format or vice versa.

About the Author

Anshuman Singh

Data Science Learner & Python Developer

Name: Anshuman Singh
Email: anshumansingh3697@gmail.com
Website: anshuman365.github.io
GitHub: github.com/anshuman365
LinkedIn: linkedin.com/in/anshuman-singh-662183303

Anshuman Singh is an enthusiastic data science learner and Python developer on a journey to master the art of data analysis and machine learning. With a passion for turning raw data into meaningful insights, he is constantly exploring new technologies and methodologies in the data science landscape.

Through hands-on projects and continuous learning, Anshuman has developed strong skills in data manipulation with pandas, data visualization, and building machine learning models. He believes in the power of practical learning and enjoys tackling real-world data challenges.

This book represents his learning journey and commitment to sharing knowledge with fellow learners. When not immersed in code and data, he enjoys contributing to open-source projects, writing about his learning experiences, and connecting with the data science community.

"The beautiful thing about learning is that nobody can take it away from you."

Keep Learning, Keep Growing
Connect with me for learning together!

Congratulations!

You've completed the Pandas Mastery journey!

You now possess comprehensive pandas skills that will serve you in all your data science endeavors. Remember, learning is a continuous journey - keep exploring, keep building, and keep growing!

What's next? Continue your journey with machine learning, deep learning, and big data technologies!

Happy Coding!