

B B C A 4 0 3

Introduction to Data Analytics using Python

MODULE 4

Data Loading, Storage & File Formats
Data Wrangling

Prof. Prashant R Kaigaddi | Department of BCA | Academic Year: 2025-26

Module 4 — Overview

What you will learn in this module

Topics Covered

- **PART A: Data Loading, Storage, and File Formats**
 - Reading and Writing Data in Text Format (CSV, JSON, XML)
 - Binary Data Formats (HDF5, Pickle)
 - Interacting with HTML and Web APIs
 - Interacting with Databases (SQLite, SQLAlchemy)
- **PART B: Data Wrangling**
 - Combining and Merging Data Sets
 - Reshaping and Pivoting
 - Data Transformation
 - String Manipulation

Learning Objectives (CO-4)

By the end of Module 4, students will be able to:

Load data

Read CSV, JSON, Excel, and HTML files into pandas DataFrames

Store data

Write processed data back to files and databases

Connect to APIs

Fetch live data from web APIs using Python

Wrangle data

Combine, merge, reshape, and pivot datasets

Transform data

Clean, filter, and transform data columns

String ops

Apply advanced string manipulation techniques

Data Loading, Storage & File Formats

Introduction

Why Data Loading Matters

- Real-world data exists in many formats: CSV, JSON, Excel, HTML, Databases
- Python's pandas library provides powerful I/O functions
- Data loading is the FIRST step in any data analytics pipeline
- Incorrect loading leads to corrupt or incomplete analysis
- pandas read_* functions auto-detect headers, data types, separators
- write_* / to_* functions help export cleaned data

Reading & Writing Data in Text Format

CSV, TSV, JSON, XML

Format	Extension	pandas Function	Description
CSV	.csv	<code>read_csv()</code> / <code>to_csv()</code>	Comma-Separated Values — most common
TSV	.tsv / .txt	<code>read_csv(sep='\t')</code>	Tab-Separated Values
JSON	.json	<code>read_json()</code> / <code>to_json()</code>	JavaScript Object Notation
Excel	.xlsx / .xls	<code>read_excel()</code> / <code>to_excel()</code>	Microsoft Excel format
HTML	.html	<code>read_html()</code>	Parses HTML tables
Fixed-Width	.txt	<code>read_fwf()</code>	Fixed-width column format

pd.read_csv() — Reading CSV Files

Most commonly used data loading function

Key Parameters

- filepath_or_buffer — path to file or URL
- sep / delimiter — column separator (default: ',')
- header — row number for column names (default: 0)
- names — custom list of column names
- index_col — column to use as row index
- usecols — select specific columns to load
- nrows — number of rows to read (useful for large files)
- na_values — values to treat as NaN (missing)

Sample CSV Content

```
name,age,city
Alice,25,Mumbai
Bob,30,Delhi
Carol,22,Pune
```

 *Tip: Always use df.head() after loading to verify data*

```
df = pd.read_csv('data.csv',
usecols=['name', 'age'])
df.head()
```

Python Program: Reading CSV Files

pd.read_csv() examples

```
import pandas as pd

# --- Basic CSV reading ---
df = pd.read_csv('students.csv')
print(df.head())          # First 5 rows
print(df.shape)           # (rows, columns)
print(df.dtypes)          # Data types of each column

# --- Read with specific columns ---
df2 = pd.read_csv('students.csv', usecols=['name', 'marks'])

# --- Custom separator (TSV) ---
df3 = pd.read_csv('data.tsv', sep='\t')

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

Output: Reading CSV Files

Expected console output

► OUTPUT

	name	age	city	marks
0	Alice	25	Mumbai	88
1	Bob	30	Delhi	75
2	Carol	22	Pune	92
3	David	28	Chennai	68
4	Eva	35	Kolkata	95

(5, 4)

```
name      object
age       int64
city      object
marks     int64
dtype: object
```


Reading & Writing JSON Files

JavaScript Object Notation

JSON — Key Concepts

- JSON stores data as key-value pairs, similar to Python dictionaries
- Widely used in Web APIs — most APIs return JSON responses
- `pd.read_json()` automatically parses JSON into DataFrame
- JSON orientation modes: 'records', 'index', 'columns', 'values'
- `to_json()` exports DataFrame to JSON format

JSON Structure

```
[  
  {"name": "Alice", "age": 25},  
  {"name": "Bob", "age": 30},  
  {"name": "Carol", "age": 22}  
]
```

*JSON is human-readable and
language-independent.
Used in: REST APIs, configs,
web data exchange.*

Python Program: JSON Files

read_json() and to_json()

```
import pandas as pd
import json

# --- Reading JSON file ---
df = pd.read_json('students.json')
print(df)

# --- Reading JSON string directly ---
json_str = '[{"name": "Alice", "marks": 88}, {"name": "Bob", "marks": 75}]'
df2 = pd.read_json(json_str)
print(df2)

# --- Writing to JSON ---
df.to_json('output.json', orient='records', indent=2)
# orient='records' → list of dicts (one per row)
```

Output: JSON Reading

Expected console output

► OUTPUT

	name	age
0	Alice	25
1	Bob	30
2	Carol	22

	name	marks
0	Alice	88
1	Bob	75

output.json content:

```
[  
  {"name": "Alice", "age": 25},  
  {"name": "Bob", "age": 30},  
  {"name": "Carol", "age": 22}  
]
```

Reading Excel Files

pd.read_excel() with openpyxl

Excel File Handling

- Install: `pip install openpyxl xlrd`
- `pd.read_excel()` reads .xlsx and .xls files
- `sheet_name` parameter: read specific sheet by name or index
- Multiple sheets can be read into a dict of DataFrames
- `ExcelWriter` allows writing multiple sheets to one file
- `header`, `skiprows`, `usecols` work like `read_csv`

Reading Single Sheet

```
df = pd.read_excel('data.xlsx',  
                  sheet_name='Sheet1')
```

Reading All Sheets

```
all_sheets = pd.read_excel(  
    'data.xlsx', sheet_name=None)
```

Writing to Excel

```
df.to_excel('out.xlsx', index=False)
```

Binary Data Formats

HDF5 and Pickle

Why Binary Formats?

- Faster to read/write compared to text formats
- Preserve data types exactly — no type conversion issues
- Ideal for large datasets and scientific computing

Pickle (.pkl)

```
import pickle
```

```
# Save
```

```
df.to_pickle('data.pkl')
```

```
# Load
```

```
df = pd.read_pickle('data.pkl')
```

✓ Python-specific binary format

✓ Very fast for pandas objects

HDF5 (.h5 / .hdf5)

```
# Save
```

```
df.to_hdf('data.h5', key='df')
```

```
# Load
```

```
df = pd.read_hdf('data.h5', 'df')
```

✓ Hierarchical Data Format

✓ Great for large numerical data

Python Program: Binary Formats

Pickle and HDF5 examples

```
import pandas as pd

# Create a sample DataFrame
df = pd.DataFrame({'name': ['Alice', 'Bob'], 'score': [88, 75]})

# ===== PICKLE =====
df.to_pickle('students.pkl')          # Save as pickle
df_loaded = pd.read_pickle('students.pkl') # Reload
print('Pickle loaded:\n', df_loaded)

# ===== HDF5 =====
# pip install tables (required for HDF5)
df.to_hdf('students.h5', key='data', mode='w') # Save
df_h5 = pd.read_hdf('students.h5', 'data')     # Load
print('HDF5 loaded:\n', df_h5)
```

Output: Binary Formats

Expected console output

► OUTPUT

Pickle loaded:

	name	score
0	Alice	88
1	Bob	75

HDF5 loaded:

	name	score
0	Alice	88
1	Bob	75

Both formats preserve dtypes exactly

Pickle is Python-only; HDF5 is cross-language

Interacting with HTML and Web APIs

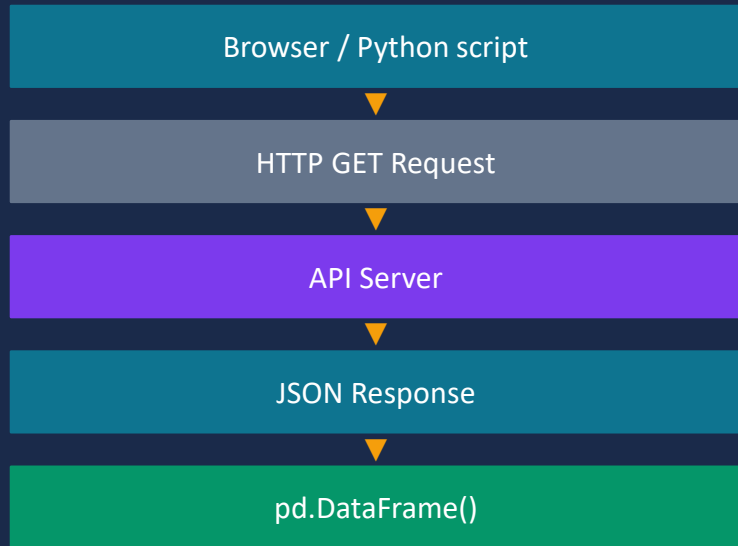
Extracting data from web sources

Web APIs with requests

Reading HTML Tables

- `pd.read_html(url)` extracts ALL tables from a webpage
- Returns a list of DataFrames — one per HTML `<table>` tag
- Works with Wikipedia, financial sites, data portals
- Requires `lxml` or `html5lib`: `pip install lxml`
- APIs provide structured data (usually JSON) via HTTP requests library: `pip install requests`
- GET request → `response.json()` → `pd.DataFrame()`
- API key / authentication may be required
- Real-world APIs: OpenWeather, CoinGecko, REST Countries

🌐 Data Flow: API → Python



Python Program: HTML & Web API

pd.read_html() and requests

```
import pandas as pd
import requests

# ===== Reading HTML Table =====
url = 'https://en.wikipedia.org/wiki/List_of_countries_by_population'
tables = pd.read_html(url)    # Returns a list of DataFrames
df = tables[0]                # First table
print(df.head())

# ===== Fetching from Web API =====
api_url = 'https://restcountries.com/v3.1/all'
response = requests.get(api_url)    # HTTP GET
data = response.json()              # Parse JSON
df_api = pd.DataFrame(data)
print(df_api[['name', 'region']].head())
```

Output: Web API Data

Expected console output

► OUTPUT

```
# HTML Table (Wikipedia countries):
```

	Country	Population	Area (km ²)
0	China	1412600000	9596960
1	India	1375586000	3287263
2	USA	331893745	9372610

```
# REST Countries API:
```

	name	region
0	Austria	Europe
1	Angola	Africa
2	Åland Islands	Europe
3	Albania	Europe

```
# response.status_code == 200 → successful request
```

Interacting with Databases

SQLite and SQLAlchemy

Connecting Python to Databases

- pandas integrates with SQL databases via `pd.read_sql()`
- SQLite: lightweight file-based database, built into Python
- SQLAlchemy: ORM toolkit for connecting to MySQL, PostgreSQL, etc.
- `pd.read_sql_query()` — execute SELECT and return DataFrame
- `df.to_sql()` — write DataFrame to a database table
- `con` parameter: database connection object

SQLite (Built-in)

```
import sqlite3  
con = sqlite3.connect('school.db')
```

SQLAlchemy

```
from sqlalchemy import create_engine  
engine = create_engine('sqlite:///school.db')
```

Query to DataFrame

```
df = pd.read_sql('SELECT * FROM students', con)
```

Python Program: Database Interaction

SQLite + pandas

```
import pandas as pd
import sqlite3

# --- Create/connect to SQLite database ---
con = sqlite3.connect('school.db')

# --- Create sample DataFrame and write to DB ---
df = pd.DataFrame({'name':['Alice','Bob','Carol'], 'marks':[88,75,92]})
df.to_sql('students', con, if_exists='replace', index=False)

# --- Read from database ---
df_read = pd.read_sql('SELECT * FROM students WHERE marks > 80', con)
print(df_read)

# --- Close connection ---
con.close()
```

Output: Database Interaction

Expected console output

► OUTPUT

```
# Writing to DB: students table created/replaced
```

```
# Reading with WHERE marks > 80:
```

	name	marks
0	Alice	88
1	Carol	92

```
# if_exists options:
```

```
# 'replace' → drop table and recreate
```

```
# 'append' → add rows to existing table
```

```
# 'fail' → raise error if table exists
```

Data Wrangling — Introduction

Preparing data for analysis

Data Wrangling = Cleaning, Transforming, and Restructuring raw data into an analysis-ready format

1. Combine

Merge & concatenate multiple datasets

2. Reshape

Pivot, melt, stack, unstack

3. Transform

Map, apply, replace, bin values

4. Clean Strings

Extract, replace, split text

Combining DataFrames — `pd.concat()`

Stacking datasets vertically or horizontally

`pd.concat()` Explained

- `pd.concat()` joins DataFrames along an axis
- `axis=0` (default) → stack rows (vertical / row-wise)
- `axis=1` → join side-by-side (column-wise)
- `ignore_index=True` → reset index after concat
- `keys` parameter → add hierarchical index to identify source
- `join='outer'` (default) keeps all columns; 'inner' keeps common only

Visual: Row Concat (`axis=0`)

df1
A 1
B 2

df2
C 3
D 4

*Use when datasets have
the same columns/schema*

▼ concat

result
A 1
B 2
C 3
D 4

Python Program: pd.concat()

Combining DataFrames

```
import pandas as pd

# Two DataFrames with same columns
df1 = pd.DataFrame({'name':['Alice','Bob'], 'marks':[88,75]})
df2 = pd.DataFrame({'name':['Carol','David'], 'marks':[92,68]})

# Row concat (axis=0) – stack vertically
result = pd.concat([df1, df2], ignore_index=True)
print(result)

# Column concat (axis=1) – side by side
df3 = pd.DataFrame({'grade':['A','B','A+','C']})
combined = pd.concat([result, df3], axis=1)
print(combined)
```


Output: pd.concat()

Expected console output

► OUTPUT

Row concat result:

	name	marks
0	Alice	88
1	Bob	75
2	Carol	92
3	David	68

Column concat result:

	name	marks	grade
0	Alice	88	A
1	Bob	75	B
2	Carol	92	A+
3	David	68	C

Merging DataFrames — `pd.merge()`

SQL-style JOIN operations

Types of Merge (JOIN)

INNER JOIN

Only matching rows in both DataFrames

LEFT JOIN

All rows from left DF + matches from right

RIGHT JOIN

All rows from right DF + matches from left

OUTER JOIN

All rows from both DFs (NaN for non-matches)

```
Syntax: merged = pd.merge(df_left, df_right, on='key_column', how='inner')
```

Python Program: pd.merge()

JOIN operations on DataFrames

```
import pandas as pd

# Students table
students = pd.DataFrame({'id':[1,2,3], 'name':['Alice','Bob','Carol']})

# Marks table
marks = pd.DataFrame({'id':[1,2,4], 'marks':[88,75,90]})

# INNER JOIN – only matching IDs
inner = pd.merge(students, marks, on='id', how='inner')
print('Inner:\n', inner)

# LEFT JOIN – all students, NaN if no marks
left = pd.merge(students, marks, on='id', how='left')
print('Left:\n', left)
```

Output: pd.merge()

Expected console output

► OUTPUT

Inner:

	id	name	marks
0	1	Alice	88
1	2	Bob	75

Left:

	id	name	marks
0	1	Alice	88.0
1	2	Bob	75.0
2	3	Carol	NaN

← Carol has no marks entry

NaN appears for non-matching rows in LEFT JOIN

Carol (id=3) is not in marks table → NaN

Reshaping — Melt & Pivot

Wide ↔ Long format conversion

Wide vs Long Format

- Wide format: each variable in a separate column (common in Excel)
- Long format: each row = one observation (better for analysis)
- `pd.melt()` → converts Wide to Long format
- `df.pivot()` → converts Long to Wide format
- `pd.pivot_table()` → powerful aggregation + reshaping
- `stack()` / `unstack()` → work with multi-level indexes

WIDE FORMAT

name	math	science
Alice	88	92
Bob	75	80

→ melt
pivot ←

LONG FORMAT

name	subject	score
Alice	math	88
Alice	science	92
Bob	math	75
Bob	science	80

Python Program: Melt & Pivot

Reshaping DataFrames

```
import pandas as pd

# Wide format DataFrame
df_wide = pd.DataFrame({'name':['Alice','Bob'],
                        'math':[88,75], 'science':[92,80]})

# MELT: Wide → Long
df_long = pd.melt(df_wide, id_vars=['name'],
                  var_name='subject', value_name='score')
print(df_long)

# PIVOT: Long → Wide
df_back = df_long.pivot(index='name', columns='subject', values='score')
df_back.reset_index(inplace=True)
print(df_back)
```

Output: Melt & Pivot

Expected console output

► OUTPUT

```
# df_long (after melt):
```

	name	subject	score
0	Alice	math	88
1	Bob	math	75
2	Alice	science	92
3	Bob	science	80

```
# df_back (after pivot – back to wide):
```

	subject	name	math	science
0		Alice	88	92
1		Bob	75	80

Data Transformation

map(), apply(), replace(), cut()

Key Transformation Functions

map()

Element-wise transformation on a Series
`df['col'].map({'A':1,'B':2})`

replace()

Replace specific values
`df.replace({'old_val':'new_val'})`

rename()

Rename columns or index
`df.rename(columns={'old':'new'})`

apply()

Apply a function along axis (row/column)
`df.apply(lambda x: x.max(), axis=0)`

cut()

Bin continuous data into categories
`pd.cut(df['age'], bins=[0,18,60,100])`

drop_duplicates()

Remove duplicate rows
`df.drop_duplicates(subset=['name'])`

Python Program: Data Transformation

map, apply, replace, cut

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'name': ['Alice', 'Bob', 'Carol', 'David'],
                   'marks': [88, 75, 45, 92], 'gender': ['F', 'M', 'F', 'M']})

# map() – encode gender
df['gender_code'] = df['gender'].map({'F': 0, 'M': 1})

# apply() – custom function per row
df['result'] = df['marks'].apply(lambda x: 'Pass' if x >= 50 else 'Fail')

# cut() – bin marks into grades
df['grade'] = pd.cut(df['marks'], bins=[0, 49, 74, 89, 100],
                    labels=['F', 'C', 'B', 'A'])

print(df)
```

Output: Data Transformation

Expected console output

► OUTPUT

	name	marks	gender	gender_code	result	grade
0	Alice	88	F	0	Pass	B
1	Bob	75	M	1	Pass	C
2	Carol	45	F	0	Fail	F
3	David	92	M	1	Pass	A

map() → replaced gender with numeric code

apply() → classified pass/fail with custom lambda

cut() → assigned letter grade based on bins

String Manipulation in pandas

Using the .str accessor

The .str Accessor

- pandas provides .str accessor for vectorized string operations
- Applies string methods element-wise on an entire Series
- Much faster than looping with for loops

Method	Purpose
str.lower() / upper()	Convert to lowercase / uppercase
str.strip() / lstrip() / rstrip()	Remove leading/trailing whitespace
str.replace(old, new)	Replace substring
str.split(sep)	Split string into list
str.contains(pattern)	Boolean mask: does string contain pattern?
str.startswith() / endswith()	Check string start/end
str.len()	Length of each string
str.extract(regex)	Extract groups using regex

Python Program: String Manipulation

pandas .str accessor methods

```
import pandas as pd

df = pd.DataFrame({'name': ['  alice ', 'BOB', 'Carol '],
                  'email': ['alice@gmail.com', 'bob@yahoo.com', 'carol@gmail.com']})

# Clean and normalize names
df['name'] = df['name'].str.strip().str.title() # strip spaces, title case

# Extract email domain
df['domain'] = df['email'].str.split('@').str[1]

# Filter: gmail users only
gmail_users = df[df['email'].str.contains('gmail')]

# Check email length
df['email_len'] = df['email'].str.len()
print(df)
print('Gmail users:\n', gmail_users)
```

Output: String Manipulation

Expected console output

► OUTPUT

	name	email	domain	email_len
0	Alice	alice@gmail.com	gmail	15
1	Bob	bob@yahoo.com	yahoo	13
2	Carol	carol@gmail.com	gmail	15

Gmail users:

	name	email	domain	email_len
0	Alice	alice@gmail.com	gmail	15
2	Carol	carol@gmail.com	gmail	15

```
# str.strip() removed extra spaces
# str.title() capitalized first letter
# str.split('@').str[1] extracted domain
```

Regular Expressions in pandas

Pattern matching with str.extract()

Using Regex with .str

- Regular expressions (regex) define search patterns for text
- `str.extract(r'pattern')` — extract captured groups
- `str.findall(r'pattern')` — find all matches (returns list)
- `str.match(r'pattern')` — True/False: does start match?
- `str.replace(r'pattern', 'new')` — regex-based replacement
- Use raw strings `r''` to avoid Python escape conflicts

Extract Phone Numbers

```
df['phone'].str.extract(r'(\d{10})')
```

Extract Year from Date

```
df['date'].str.extract(r'(\d{4})')
```

Find Words

```
df['text'].str.findall(r'\b\w{5}\b')
```

Python Program: Regex in String Manipulation

str.extract and str.replace

```
import pandas as pd

df = pd.DataFrame({'info': ['Alice: 9876543210 (Mumbai)',
                           'Bob: 8765432109 (Delhi)',
                           'Carol: 7654321098 (Pune)']})

# Extract name (word before ':')
df['name'] = df['info'].str.extract(r'^(\w+)')

# Extract 10-digit phone number
df['phone'] = df['info'].str.extract(r'(\d{10})')

# Extract city (word inside parentheses)
df['city'] = df['info'].str.extract(r'\(((^)+)\)')

print(df[['name', 'phone', 'city']])
```

Output: Regex String Extraction

Expected console output

► OUTPUT

	name	phone	city
0	Alice	9876543210	Mumbai
1	Bob	8765432109	Delhi
2	Carol	7654321098	Pune

Regex patterns used:

`r'^(\w+)'` → word at start of string

`r'(\d{10})'` → exactly 10 digits

`r'\(([^\)]+)\)'` → text inside parentheses

`str.extract()` returns a DataFrame with one column per group

Real-World Applications

Where these techniques are used in industry

E-Commerce Analytics

Load sales data from CSV/Excel, merge with inventory DB, pivot by region/category

Financial Analysis

Fetch stock data via API (yfinance), load historical prices, transform and visualize

Healthcare Systems

Read patient records from hospital DB, merge with treatment tables, transform diagnosis codes

Social Media Mining

Fetch tweets/posts via API, extract hashtags/mentions using regex, pivot by sentiment

Government Data

Read open datasets from data.gov.in in CSV/JSON, merge multiple years, analyze trends

HR Analytics

Load employee data from Excel, merge with payroll DB, apply transformations to compute KPIs

Complete Data Pipeline: Load → Wrangle → Export

End-to-end example

```
import pandas as pd
import sqlite3

# STEP 1: Load CSV data
df = pd.read_csv('employees.csv')

# STEP 2: Clean strings
df['name'] = df['name'].str.strip().str.title()
df['dept'] = df['dept'].str.lower().str.replace(' ', '_')

# STEP 3: Transform – add category column
df['level'] = pd.cut(df['salary'],
                    bins=[0,30000,60000,100000],
                    labels=['Junior', 'Mid', 'Senior'])

# STEP 4: Export to SQLite database
con = sqlite3.connect('company.db')
df.to_sql('employees', con, if_exists='replace', index=False)
print('Pipeline complete! Rows saved:', len(df))
```

MODULE 4 — SUMMARY

Data Loading

`read_csv`, `read_json`, `read_excel`, `read_html`, `read_sql`

Web & APIs

`read_html` for tables; `requests` + `pd.DataFrame` for APIs

Combining

`pd.concat` (row/col stack), `pd.merge` (JOIN operations)

Transform

`map`, `apply`, `replace`, `cut/qcut`, `rename`, `drop_duplicates`

Binary Formats

`to_pickle`, `read_pickle`, `to_hdf`, `read_hdf`

Databases

`sqlite3` / `SQLAlchemy` + `read_sql`, `to_sql`

Reshaping

`pd.melt` (wide→long), `df.pivot` (long→wide)

Strings

`.str` accessor: `strip`, `lower`, `split`, `contains`, `extract` (regex)

Key Takeaways

Most important concepts from Module 4

- 1. pandas is the Swiss Army Knife** A single library handles loading, storing, combining, and transforming data across virtually all common formats.
- 2. Always verify loaded data** Use `df.head()`, `df.info()`, `df.describe()` immediately after loading to catch issues early.
- 3. Choose the right JOIN** Use inner for only matching records, left to keep all source records, outer to keep everything.
- 4. Wide vs Long format matters** Visualization libraries prefer long format; pivot tables prefer wide — know when to melt and when to pivot.
- 5. The `.str` accessor is powerful** Never loop through strings manually — `.str` methods are vectorized and dramatically faster.

Exam-Oriented Questions

Practice for CIE and SEE

Short Answer / Theory Questions

- 1. What is the difference between `pd.merge()` and `pd.concat()`? Explain with examples.
- 2. List 5 key parameters of `pd.read_csv()` with their purpose.
- 3. Explain the difference between INNER, LEFT, RIGHT, and OUTER joins.
- 4. What is the purpose of `pd.melt()`? When is it used over `df.pivot()`?
- 5. What is the `.str` accessor in pandas? List any 4 string methods with syntax.
- 6. Explain how to read data from a SQLite database using pandas.

Programming Practice Questions

Hands-on coding exercises

Practice Question 1

Load the following two CSV files:

students.csv → [id, name, class]

marks.csv → [id, subject, marks]

Perform a LEFT JOIN on 'id'. Then:

- Filter only students with marks > 70
- Add a 'grade' column using `pd.cut()` with bins [0,49,69,84,100] → labels ['F','C','B','A']
- Export the result to output.csv

Practice Question 2

Given a DataFrame with a 'contact' column containing entries like:

'Name: Alice | Phone: 9876543210 | City: Mumbai'

Using the `.str` accessor and regular expressions:

- Extract Name, Phone, and City into separate columns
- Convert all names to title case
- Filter rows where city is 'Mumbai' or 'Delhi'
- Write the cleaned data to a SQLite database table called 'contacts'

References & Resources



Textbook

Python for Data Analysis — Wes McKinney, O'Reilly
ISBN: 978-1-449-31979-3 | Chapters 6, 7, 8



pandas Docs

<https://pandas.pydata.org/docs/>
Official reference for all I/O, merge, reshape, and string APIs



requests Library

<https://docs.python-requests.org/>
HTTP for Humans — Web API integration



SQLAlchemy

<https://www.sqlalchemy.org/>
Database connection engine for Python



Practice

Kaggle Datasets — <https://www.kaggle.com/datasets>
Real-world CSV/JSON data for hands-on practice