

BBCA403

Introduction to Data Analytics Using Python

MODULE 4

Chapter 1: Data Loading, Storage & File Formats

Chapter 2: Data Wrangling — Clean, Transform, Merge, Reshape

Chapter 1: Data Loading, Storage, and File Formats:

Reading and Writing Data in Text Format, Binary Data Formats, Interacting with HTML and Web APIs, Interacting with Databases.

Chapter 2 Data Wrangling:

Combining and Merging Data Sets, Reshaping and Pivoting, Data Transformation, String Manipulation.

CHAPTER 1: Data Loading, Storage, and File Formats

The tools in data analysis are of little use if you cannot easily import and export data in Python. pandas provides a rich, high-level interface for reading and writing tabular data to/from many formats including CSV, Excel, JSON, HDF5, SQL databases, and web APIs. This chapter covers all major input/output mechanisms used in real-world data analysis.

Input/Output in pandas — Four Main Categories

- Reading text files (CSV, TSV, fixed-width) and other on-disk formats
- Loading data from relational and NoSQL databases
- Interacting with network sources like web APIs (JSON/XML)
- Binary formats like HDF5, Excel, and Pickle for high-performance storage

1.1 Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax, intuitive data structures, and convenient features. pandas provides several functions for reading tabular data as a DataFrame. The two most commonly used are `read_csv()` and `read_table()`.

Overview of pandas Parsing Functions

Function	Default Delimiter	Description
<code>pd.read_csv()</code>	Comma (,)	Read delimited data from file, URL, or file-like object. Most widely used.
<code>pd.read_table()</code>	Tab (\t)	Read general delimited text with any separator using <code>sep=</code> argument.
<code>pd.read_fwf()</code>	Fixed-width	Read data in fixed-width column format (no delimiters).
<code>pd.read_clipboard()</code>	(from clipboard)	Reads data from system clipboard — useful for web tables.

One of the most important features of pandas parsing functions is automatic type inference. You don't have to specify which columns are numeric, integer, boolean, or string — pandas figures it out:

- Integer columns → int64
- Decimal numbers → float64
- True/False values → bool
- Dates (if `parse_dates=True`) → datetime64
- Everything else → object (string)

NOTE

Type inference happens automatically. However, for custom date formats or special conversions, use the `converters=` or `parse_dates=` parameters.

Five Categories of Options in `read_csv` / `read_table`

Category	Parameters	Purpose
Indexing	<code>index_col</code> , <code>header</code> , <code>names</code>	Set which columns/rows are used as index or column names
Type Inference	<code>converters</code> , <code>dtype</code> , <code>na_values</code>	Control type detection and missing value handling
Date Parsing	<code>parse_dates</code> , <code>dayfirst</code> , <code>date_parser</code>	Parse dates from one or multiple columns
Iteration	<code>chunksize</code> , <code>iterator</code> , <code>nrows</code>	Process very large files in pieces
Unclean Data	<code>skiprows</code> , <code>skip_footer</code> , <code>comment</code> , <code>thousands</code>	Handle malformed, commented, or irregular data

► Example 1 — Basic CSV Reading

Suppose `ex1.csv` contains:

File: `ex1.csv`

```
a,b,c,d,message  
1,2,3,4,hello  
5,6,7,8,world  
9,10,11,12,foo
```

Python Code

```
import pandas as pd

# Method 1: read_csv (default comma separator)
df = pd.read_csv('ex1.csv')
print(df)

# Method 2: read_table with explicit separator
df2 = pd.read_table('ex1.csv', sep=',')
print(df2)
```

Output:

```
   a  b  c  d message
0  1  2  3  4  hello
1  5  6  7  8  world
2  9 10 11 12   foo
```

► Example 2 — File Without a Header Row**File: ex2.csv (no header)**

```
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Python Code

```
# Option A: Let pandas assign default names (X.1, X.2, ...)
df = pd.read_csv('ex2.csv', header=None)

# Option B: Provide custom column names
df = pd.read_csv('ex2.csv', names=['a','b','c','d','message'])
print(df)
```

```
   a  b  c  d message
0  1  2  3  4  hello
1  5  6  7  8  world
2  9 10 11 12   foo
```

► **Example 3 — Setting a Column as Index**

Python Code

```
names = ['a', 'b', 'c', 'd', 'message']

# Use 'message' column as the row index
df = pd.read_csv('ex2.csv', names=names, index_col='message')
print(df)
```

```
      a  b  c  d
message
hello  1  2  3  4
world  5  6  7  8
foo    9 10 11 12
```

► **Example 4 — Hierarchical (Multi-level) Index from Multiple Columns**

File: csv_mindex.csv

```
key1,key2,value1,value2
one,a,1,2
one,b,3,4
two,a,9,10
two,b,11,12
```

Python Code

```
# Create a hierarchical index using both key1 and key2
parsed = pd.read_csv('csv_mindex.csv', index_col=['key1','key2'])
print(parsed)
```

```
      value1 value2
key1 key2
one  a      1     2
     b      3     4
two  a      9    10
     b     11    12
```

► Example 5 — Using Regex as Delimiter (Whitespace-Separated)

Some files use variable whitespace as a separator. Use a regex with `read_table`:

File: ex3.txt (whitespace separated)

```
A B C
aaa -0.264 -1.026
bbb 0.927 0.302
ccc -0.264 -0.386
```

Python Code

```
# \s+ means one or more whitespace characters
result = pd.read_table('ex3.txt', sep='\s+')
print(result)
# pandas infers the first column is the index because
# there is one fewer column name than data columns
```

```
      A      B      C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
```

► Example 6 — Skipping Rows and Comments

File: ex4.csv (with comment rows)

```
# hey!
a,b,c,d,message
# just a comment
# another comment
1,2,3,4,hello
5,6,7,8,world
```

Python Code

```
# Skip rows 0, 2, and 3 (the comment lines)
df = pd.read_csv('ex4.csv', skiprows=[0, 2, 3])
print(df)
```

```
  a  b  c  d message
0  1  2  3  4  hello
1  5  6  7  8  world
```

► Example 7 — Handling Missing Values

Missing data handling is one of the most important features of `read_csv`. By default, pandas recognises: NA, NaN, NULL, N/A, empty string, and several others as missing.

File: ex5.csv

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
```

Python Code

```
result = pd.read_csv('ex5.csv')
print(result)

# Check what is null
print(pd.isnull(result))

# Add custom NA strings
result2 = pd.read_csv('ex5.csv', na_values=['NULL', 'missing', 'N/A'])

# Per-column NA values using a dict
result3 = pd.read_csv('ex5.csv', na_values={'message': ['foo'], 'something': ['two']})
```

```
something a  b   c d message
0   one  1  2  3.0 4   NaN
1   two  5  6  NaN 8  world
2  three  9 10 11.0 12  foo
```

Complete Parameter Reference for `read_csv` / `read_table`

Parameter	Type	Default	Description
path	str	(required)	File path, URL, or file-like object
sep / delimiter	str/regex	','	Column separator — can be a regex like <code>\s+</code>
header	int/None	0	Row number for column names. None if no header.
index_col	int/str/list	None	Column(s) to use as row index. List = hierarchical index.
names	list	None	Custom column names. Use with <code>header=None</code> .

Parameter	Type	Default	Description
skiprows	list/int	None	Row numbers or count to skip from beginning.
skip_footer	int	0	Number of lines to skip from end of file.
na_values	list/dict	(built-in)	Additional strings to treat as NaN. Dict = per-column.
comment	str	None	Character to indicate start of comment. Rest of line ignored.
parse_dates	bool/list	False	Parse date columns. True = try all. List = specific columns.
dayfirst	bool	False	For ambiguous dates, treat first part as day (DD/MM).
date_parser	function	None	Custom function to parse dates.
nrows	int	None	Read only this many rows from the beginning.
chunksize	int	None	Return TextParser for iteration over chunks.
iterator	bool	False	Return TextParser object for incremental reading.
encoding	str	None	Text encoding: 'utf-8', 'latin1', 'utf-16', etc.
converters	dict	None	Dict of {col: function} for custom conversions.
thousands	str	None	Thousands separator (e.g., ',').
squeeze	bool	False	If only 1 data column, return Series instead of DataFrame.
verbose	bool	False	Print parser info (e.g., number of missing values placed).

Reading Text Files in Pieces — Large File Handling

When working with very large files (millions of rows), reading the entire file into memory may not be feasible. pandas provides two strategies:

► Strategy 1 — nrows: Read Only First N Rows

Python Code

```
# Read only the first 5 rows for inspection
df_sample = pd.read_csv('large_file.csv', nrows=5)
print(df_sample)
```



```
# Useful for: previewing structure, testing parsing options
```

► **Strategy 2 — chunksize: Iterate in Chunks**

Python Code

```
chunker = pd.read_csv('ex6.csv', chunksize=1000)
# chunker is a TextParser object

# Example: Aggregate key column frequencies across chunks
tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
print(tot[:10])

# Output shows top keys across entire 10,000-row file:
# E    368
# X    364
# L    346 ...
```

**TIP**

TextParser also has a `get_chunk(n)` method to read exactly `n` rows at a time, giving fine-grained control over reading.

Writing Data to Text Format

Once you have processed data in pandas, you can write it back to disk using `to_csv()` on both DataFrames and Series.

► **DataFrame.to_csv() — All Options**

Python Code

```
import pandas as pd, sys

data = pd.read_csv('ex5.csv')

# Write to file (with row index)
data.to_csv('output.csv')

# Print to screen (stdout)
data.to_csv(sys.stdout)
```

```
# Use custom delimiter
data.to_csv(sys.stdout, sep='|')

# Fill NaN with a visible string
data.to_csv(sys.stdout, na_rep='NULL')

# Suppress row index and column headers
data.to_csv(sys.stdout, index=False, header=False)

# Write only selected columns
data.to_csv(sys.stdout, index=False, columns=['a','b','c'])
```

Writing a Series to CSV:

Python Code

```
import numpy as np
dates = pd.date_range('1/1/2000', periods=7)
ts = pd.Series(np.arange(7), index=dates)
ts.to_csv('timeseries.csv')

# Reading it back
ts_loaded = pd.Series.from_csv('timeseries.csv', parse_dates=True)
```

Manually Working with Delimited Formats — Python csv Module

For files with complex quoting, unusual delimiters, or malformed rows that confuse `read_csv`, Python's built-in `csv` module gives manual control:

Python Code

```
import csv

# Open and create a reader
f = open('ex7.csv')
reader = csv.reader(f)

# Iterate: each row is a list of strings
for line in reader:
    print(line)
# ['a', 'b', 'c']
# ['1', '2', '3']
# ['1', '2', '3', '4'] <- malformed row!
```

```
# Build a dict from header + value rows
lines = list(csv.reader(open('ex7.csv')))
header, values = lines[0], lines[1:]
data_dict = {h: v for h, v in zip(header, zip(*values))}
print(data_dict) # {'a': ('1','1'), 'b': ('2','2'), 'c': ('3','3')}
```

► Defining a Custom CSV Dialect

Python Code

```
# Define custom format rules by subclassing csv.Dialect
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)

# Or pass options directly without subclassing:
reader = csv.reader(f, delimiter='|')

# Writing with csv.writer
with open('output.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(['one', 'two', 'three'])
    writer.writerow(['1', '2', '3'])
```

csv.Dialect Option	Default	Description
delimiter	','	Character separating fields
lineterminator	'\r\n'	Line terminator for writing
quotechar	"	Quote character for fields with special characters
quoting	QUOTE_MINIMAL	QUOTE_ALL, QUOTE_MINIMAL, QUOTE_NONNUMERIC, or QUOTE_NONE
skipinitialspace	False	Ignore whitespace after each delimiter
doublequote	True	How to handle quoting char inside a field — doubled if True
escapechar	None	Escape character when quoting=QUOTE_NONE

JSON Data

JSON (JavaScript Object Notation) has become one of the most common formats for exchanging data via web APIs and between applications. It is far more flexible than CSV — it supports nested objects, arrays, and null values.

KEY CONCEPT

JSON supports these types: objects (dicts), arrays (lists), strings, numbers, booleans, and null. The only difference from Python: JSON uses null (not None) and true/false (lowercase).

► Reading and Writing JSON

Python Code

```
import json

# A JSON string
obj = '''
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
               {"name": "Katie", "age": 33, "pet": "Cisco"}]}
'''

# Parse JSON string to Python dict
result = json.loads(obj)
print(type(result)) # <class 'dict'>
print(result['name']) # 'Wes'

# Convert Python dict back to JSON string
asjson = json.dumps(result)
print(type(asjson)) # <class 'str'>

# Build DataFrame from nested JSON list
siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])
print(siblings)
```

```
  name  age
0  Scott  25
1  Katie  33
```

XML Parsing with lxml.objectify

XML (Extensible Markup Language) is used widely in government data, transportation, healthcare, and enterprise systems. Here we use lxml.objectify to parse XML records into a DataFrame.

Python Code

```
# Example XML record (MTA Performance Data):
# <INDICATOR>
# <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
# <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
# <PERIOD_YEAR>2011</PERIOD_YEAR>
# <PERIOD_MONTH>12</PERIOD_MONTH>
# <YTD_TARGET>97.00</YTD_TARGET>
# <YTD_ACTUAL>96.50</YTD_ACTUAL>
# </INDICATOR>

from lxml import objectify
import pandas as pd

parsed = objectify.parse(open('Performance_MNR.xml'))
root = parsed.getroot()

# Fields to skip
skip_fields =
['PARENT_SEQ', 'INDICATOR_SEQ', 'DESIRED_CHANGE', 'DECIMAL_PLACES']

# Build list of dicts from XML records
data = []
for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)

# Convert to DataFrame
perf = pd.DataFrame(data)
print(perf.head())
```

1.2 Binary Data Formats

Binary formats store data in raw binary encoding rather than human-readable text. They are significantly faster to read/write and more compact than CSV, making them ideal for large datasets.

Format	Extension	Library	Speed	Portability	Best For
Pickle	.pkl / .pickle	Python built-in	★ ★ ★ ★	Python only	Short-term Python caching
HDF5	.h5 / .hdf5	PyTables / h5py	★ ★ ★ ★ ★	Cross-language	Very large datasets, time series
Excel	.xlsx / .xls	openpyxl/xlrd	★ ★	Microsoft Office	Reports, sharing non-technical data
Parquet	.parquet	pyarrow/fastparquet	★ ★ ★ ★ ★	Cross-language	Big data, columnar analytics
Feather	.feather	pyarrow	★ ★ ★ ★ ★	Python/R	Fast interchange between Python/R

Pickle — Python's Built-in Serialization

pickle is Python's native object serialization protocol. Every pandas object supports it via `to_pickle()` and `pd.read_pickle()`.

Python Code

```
import pandas as pd

frame = pd.read_csv('ex1.csv')
print(frame)
# a b c d message
# 0 1 2 3 4 hello ...

# Save as pickle binary file
frame.to_pickle('frame_pickle')

# Load pickle back
loaded = pd.read_pickle('frame_pickle')
print(loaded)
```



Pickle is recommended only for short-term storage. It is hard to guarantee format stability across different versions of libraries. For long-term or cross-system use, prefer HDF5, Parquet, or CSV.

HDF5 — High-Performance Data Format

HDF5 (Hierarchical Data Format version 5) is a C library with interfaces in Python, Java, MATLAB, and R. It provides:

- Hierarchical file-system-like structure inside one file
- On-the-fly compression (gzip, lzf, blosc)
- Efficient reading of subsets (slices) of large arrays
- Both fixed format (fastest) and table format (queryable)

pandas wraps HDF5 through the HDFStore class (which uses PyTables internally):

Python Code

```
import pandas as pd
import numpy as np

frame = pd.DataFrame({'a': np.random.randn(100)})

# --- WRITING ---
store = pd.HDFStore('mydata.h5')
store['obj1'] = frame          # Store entire DataFrame
store['obj1_col'] = frame['a'] # Store a single column (Series)

# View what is stored
print(store)
# <class HDFStore> File path: mydata.h5
# obj1    DataFrame
# obj1_col Series

# --- READING ---
retrieved = store['obj1']
print(retrieved.head())

# Close the store
store.close()

# --- Shortcut methods ---
```

```
frame.to_hdf('mydata.h5', 'df', format='table')
loaded = pd.read_hdf('mydata.h5', 'df', where=['index < 10'])
```

**WARNING**

HDF5 is not a database — it is best suited for write-once, read-many datasets. Concurrent writes from multiple processes can corrupt the file.

Reading and Writing Excel Files

pandas supports Excel files via ExcelFile (for reading) and ExcelWriter (for writing), which use xlrd and openpyxl under the hood.

Python Code

```
import pandas as pd

# ===== READING =====

# Method 1: ExcelFile object (gives access to sheet names)
xls = pd.ExcelFile('data.xlsx')
print(xls.sheet_names) # ['Sheet1', 'Sheet2', 'Summary']

# Parse a specific sheet
df = xls.parse('Sheet1')

# Method 2: Direct shortcut
df = pd.read_excel('data.xlsx', sheet_name='Sheet1', index_col=0)

# ===== WRITING =====

# Method 1: ExcelWriter (write multiple sheets)
writer = pd.ExcelWriter('output.xlsx')
df.to_excel(writer, sheet_name='Results')
df.describe().to_excel(writer, sheet_name='Summary')
writer.save()

# Method 2: One-liner (single sheet)
df.to_excel('output.xlsx', sheet_name='Data', index=False)
```


1.3 Interacting with HTML and Web APIs

Many websites expose data in HTML tables or JSON APIs. Python provides excellent tools for both: lxml/BeautifulSoup for HTML scraping, and the requests library for web API calls.

HTML Scraping with lxml

The lxml library provides a fast, memory-efficient HTML/XML parser. We can extract tables from any webpage using its findall() and XPath querying:

Python Code

```
from lxml.html import parse
from urllib2 import urlopen # Python 2; use urllib.request in Python 3
import pandas as pd

# Step 1: Parse the HTML page
parsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL'))
doc = parsed.getroot()

# Step 2: Extract all hyperlinks
links = doc.findall('.//a')
urls = [lnk.get('href') for lnk in links if lnk.get('href')]
print(urls[-5:])

# Step 3: Get the display text of a link
lnk = links[28]
print(lnk.get('href')) # URL
print(lnk.text_content()) # Visible text

# Step 4: Find all tables
tables = doc.findall('.//table')

# Step 5: Extract specific table (trial and error to find index)
calls_table = tables[9]
puts_table = tables[13]
```

► Converting an HTML Table to a DataFrame

Python Code

```
# Helper: extract text from td or th cells
def _unpack(row, kind='td'):
    elts = row.findall('./%s' % kind)
    return [val.text_content() for val in elts]

# Full parser using pandas TextParser for type inference
from pandas.io.parsers import TextParser

def parse_options_data(table):
    rows = table.findall('./tr')
    header = _unpack(rows[0], kind='th')
    data = [_unpack(r) for r in rows[1:]]
    return TextParser(data, names=header).get_chunk()

# Build DataFrames
call_data = parse_options_data(calls_table)
put_data = parse_options_data(puts_table)

print(call_data[:5])
```

	Strike	Symbol	Last Chg	Bid	Ask	Vol	Open	Int		
0	295	AAPL120818C00295000	310.40	0.0	289.80	290.80	1	169		
1	300	AAPL120818C00300000	277.10	1.7	284.80	285.80	2	478		

Interacting with Web APIs (REST/JSON APIs)

Web APIs return structured data — usually JSON. We send an HTTP request and parse the response. The requests library simplifies this greatly:

Python Code

```
import requests, json
import pandas as pd

# HTTP GET request to Twitter Search API
url = 'http://search.twitter.com/search.json?q=python+pandas'
resp = requests.get(url)

# Check HTTP status
print(resp.status_code) # 200 = success

# Parse the JSON response body
data = json.loads(resp.text)
```

```
# Inspect response structure
print(data.keys())
# ['results', 'max_id', 'since_id', 'query', 'page', ...]

# 'results' is a list of tweet dicts
print(data['results'][0].keys())
# ['created_at', 'from_user', 'id', 'text', 'geo', ...]

# Extract specific fields into a DataFrame
tweet_fields = ['created_at', 'from_user', 'id', 'text']
tweets = pd.DataFrame(data['results'], columns=tweet_fields)
print(tweets)
```

```
   created_at  from_user   id      text
0 Mon, 25 Jun 2012... wesmckinn 220230... pandas is amazing!
1 Mon, 25 Jun 2012... datanerd 220240... read_csv is fast...
```

1.4 Interacting with Databases

In most production data environments, data is stored in relational databases like MySQL, PostgreSQL, or SQLite. pandas provides convenient methods to load SQL query results directly into DataFrames.

SQLite — Using Python's Built-in sqlite3 Module

Python Code

```
import sqlite3
import pandas as pd

# Create in-memory SQLite database (or use file path)
con = sqlite3.connect(':memory:')

# Create a table
create_query = """
CREATE TABLE cities (
    name  VARCHAR(20),
    state VARCHAR(20),
    pop   REAL,
    rank  INTEGER
);"""
con.execute(create_query)
```

```
# Insert rows
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.60, 3),
        ('Sacramento', 'California', 1.70, 5)]

con.executemany('INSERT INTO cities VALUES(?,?,?,?)', data)
con.commit()

# Execute a SELECT query
cursor = con.execute('SELECT * FROM cities')
rows = cursor.fetchall()

# Get column names from cursor description
cols = [x[0] for x in cursor.description]

# Build DataFrame manually
df = pd.DataFrame(rows, columns=cols)
print(df)
```

```
   name  state  pop rank
0  Atlanta  Georgia  1.25   6
1 Tallahassee  Florida  2.60   3
2  Sacramento  California  1.70   5
```

pd.read_sql() — The Convenient Way

Instead of manually fetching rows and building a DataFrame, use `pd.read_sql()` which does both in one step:

Python Code

```
# Read entire table
df = pd.read_sql('SELECT * FROM cities', con)

# Filtered query
df_big = pd.read_sql('SELECT name, pop FROM cities WHERE rank > 4', con)

# Write DataFrame back to SQL
df.to_sql('cities_backup', con, if_exists='replace', index=False)

# if_exists options: fail (default), replace, append

con.close()
```

pd.read_sql() Parameter	Description
sql	SQL query string or table name to read
con	Database connection object (sqlite3, SQLAlchemy, etc.)
index_col	Column(s) to use as the row index
parse_dates	Columns to parse as dates
columns	List of columns to read (if sql is a table name)

MongoDB — NoSQL Document Database

MongoDB stores data as JSON-like documents (BSON). pymongo is the official Python driver:

Python Code

```
import pymongo
import pandas as pd

# Connect to local MongoDB (default port 27017)
con = pymongo.MongoClient('localhost', port=27017)

# Access database and collection
db = con.mydb
tweets = db.tweets

# Insert a document
tweets.insert_one({'user': 'alice', 'text': 'Hello!', 'likes': 10})
tweets.insert_one({'user': 'bob', 'text': 'pandas rocks', 'likes': 3})

# Query: find tweets with likes > 5
cursor = tweets.find({'likes': {'$gt': 5}})

# Convert result to DataFrame
results = list(cursor)
df = pd.DataFrame(results)

# Drop MongoDB internal _id column if not needed
df = df.drop('_id', axis=1)
print(df)
```

```
user text likes
0 alice Hello! 10
```

CHAPTER 2: Data Wrangling — Clean, Transform, Merge, Reshape

Much of the programming work in data analysis is spent on data preparation: loading, cleaning, transforming, and rearranging. pandas provides a high-level, flexible, and high-performance set of core manipulations to wrangle data into the right form. This chapter covers all major data wrangling techniques.

Three Core Combining Operations in pandas

- `pd.merge()` — connects rows in DataFrames based on one or more keys (like SQL JOIN)
- `pd.concat()` — glues or stacks objects together along an axis (rows or columns)
- `combine_first()` — splices overlapping data to fill in missing values from another object

2.1 Combining and Merging Data Sets

Database-style Merges with `pd.merge()`

The `merge()` function is the main entry point for combining DataFrames by linking rows using shared key column(s). This is equivalent to SQL JOINS. Understanding merge behavior is fundamental to data analysis.

► Many-to-One Merge (Most Common)

Multiple rows in the left DataFrame match a single row in the right DataFrame:

Python Code

```
import pandas as pd

df1 = pd.DataFrame({'key': ['b','b','a','c','a','a','b'], 'data1': range(7)})
df2 = pd.DataFrame({'key': ['a','b','d'], 'data2': range(3)})

# Inner join (default) — only matching keys kept
result = pd.merge(df1, df2, on='key')
print(result)
```

```
  data1 key  data2
0     2  a     0
1     4  a     0
```

```
2  5  a  0
3  0  b  1
4  1  b  1
5  6  b  1
```

NOTE

Notice: key 'c' from df1 and 'd' from df2 are missing — because merge uses inner join by default, keeping only rows where key exists in BOTH DataFrames.

► Many-to-Many Merge (Cartesian Product)

When both sides have duplicate key values, merge creates all combinations (Cartesian product):

Python Code

```
df1 = pd.DataFrame({'key': ['b','b','a','c','a','b'], 'data1': range(6)})
df2 = pd.DataFrame({'key': ['a','b','a','b','d'], 'data2': range(5)})

# Left join: all df1 rows + matching df2 rows
result = pd.merge(df1, df2, on='key', how='left')
print(result)
# 3 b rows in df1 × 2 b rows in df2 = 6 b rows in result!
```

```
data1 key data2
0    2  a  0.0
1    2  a  2.0
2    4  a  0.0
3    4  a  2.0
4    0  b  1.0
5    0  b  3.0
...  (c → NaN)
```

► Four Types of Joins — Complete Guide

Join Type	how=	Rows Kept	Missing Values
Inner Join	'inner'	Only rows where key exists in BOTH DataFrames	None (no NaN introduced)
Left Join	'left'	All rows from LEFT + matching from right	NaN in right columns where no match
Right Join	'right'	All rows from RIGHT + matching from left	NaN in left columns where no match

Join Type	how=	Rows Kept	Missing Values
Outer Join	'outer'	ALL rows from BOTH DataFrames (union)	NaN on both sides where no match

Python Code

```
df1 = pd.DataFrame({'key': ['a','b','c'], 'val1': [1,2,3]})
df2 = pd.DataFrame({'key': ['b','c','d'], 'val2': [4,5,6]})

# INNER: only b and c
print(pd.merge(df1, df2, on='key', how='inner'))

# LEFT: a, b, c (d from df2 dropped)
print(pd.merge(df1, df2, on='key', how='left'))

# RIGHT: b, c, d (a from df1 dropped)
print(pd.merge(df1, df2, on='key', how='right'))

# OUTER: a, b, c, d (all keys, NaN fills gaps)
print(pd.merge(df1, df2, on='key', how='outer'))
```

► Merging on Multiple Keys

Python Code

```
left = pd.DataFrame({'key1': ['foo','foo','bar'],
                      'key2': ['one','two','one'],
                      'lval': [1, 2, 3]})

right = pd.DataFrame({'key1': ['foo','foo','bar','bar'],
                      'key2': ['one','one','one','two'],
                      'rval': [4, 5, 6, 7]})

# Merge on both key1 AND key2 — outer join
result = pd.merge(left, right, on=['key1','key2'], how='outer')
print(result)
```

```
key1 key2 lval rval
0 bar one 3.0 6.0
1 bar two NaN 7.0
2 foo one 1.0 4.0
3 foo one 1.0 5.0
4 foo two 2.0 NaN
```


► Merging on Different Column Names**Python Code**

```
df3 = pd.DataFrame({'lkey': ['b','b','a','c'], 'data1': range(4)})
df4 = pd.DataFrame({'rkey': ['a','b','d'], 'data2': range(3)})

# Specify different key names for left and right
result = pd.merge(df3, df4, left_on='lkey', right_on='rkey')
print(result)
```

► Handling Overlapping Column Names with suffixes**Python Code**

```
left = pd.DataFrame({'key1': ['foo','bar'], 'key2': ['one','one'], 'lval': [1,3]})
right = pd.DataFrame({'key1': ['foo','bar'], 'key2': ['one','two'], 'rval': [4,6]})

# Without suffixes: overlapping key2 becomes key2_x and key2_y
r1 = pd.merge(left, right, on='key1')
print(r1.columns.tolist())
# ['key1', 'key2_x', 'lval', 'key2_y', 'rval']

# With custom suffixes
r2 = pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
print(r2.columns.tolist())
# ['key1', 'key2_left', 'lval', 'key2_right', 'rval']
```

► Merging on Index**Python Code**

```
left1 = pd.DataFrame({'key': ['a','b','a','b','c'], 'value': range(5)})
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a','b'])

# Use right DataFrames index as the join key
result = pd.merge(left1, right1, left_on='key', right_index=True)
print(result)

# Outer join to also include unmatched rows
result_outer = pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

```
# Both indexes as keys
left2 = pd.DataFrame([[1,2],[3,4],[5,6]], index=['a','c','e'],
columns=['Ohio','Nevada'])
right2 = pd.DataFrame([[7,8],[9,10],[11,12]], index=['b','c','e'],
columns=['Missouri','Alabama'])

result2 = pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
print(result2)
```

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
e	5.0	6.0	11.0	12.0

► DataFrame.join() — Merge-by-Index Shortcut

Python Code

```
# join() is a convenient method for index-based merges
result = left2.join(right2, how='outer')

# Join multiple DataFrames at once
another = pd.DataFrame([[7,8],[9,10],[11,12]], index=['a','c','e'],
columns=['NY','Oregon'])
result_multi = left2.join([right2, another])
print(result_multi)
```

Concatenating Along an Axis with pd.concat()

concat() is for stacking (appending) pandas objects along a given axis. Unlike merge (which aligns on keys), concat aligns on index labels.

► Concatenating Series — Vertical (axis=0)

Python Code

```
s1 = pd.Series([0, 1], index=['a','b'])
s2 = pd.Series([2, 3, 4], index=['c','d','e'])
s3 = pd.Series([5, 6], index=['f','g'])
```

```
# Default: stack along rows (axis=0)
result = pd.concat([s1, s2, s3])
print(result)
```

```
a  0
b  1
c  2
d  3
e  4
f  5
g  6
dtype: int64
```

► **Concatenating Series — Horizontal (axis=1)**

Python Code

```
# Stack side by side as DataFrame columns
result = pd.concat([s1, s2, s3], axis=1)
print(result)
# NaN appears where index labels do not overlap
```

```
   0  1  2
a  0.0 NaN NaN
b  1.0 NaN NaN
c  NaN 2.0 NaN
d  NaN 3.0 NaN
e  NaN 4.0 NaN
f  NaN NaN 5.0
g  NaN NaN 6.0
```

► **join= Parameter: inner vs outer**

Python Code

```
s4 = pd.concat([s1 * 5, s3])

# Outer (default): union of indexes
print(pd.concat([s1, s4], axis=1))

# Inner: intersection only (shared indexes)
print(pd.concat([s1, s4], axis=1, join='inner'))
```

► **keys= Parameter: Create Hierarchical Index**

Python Code

```
# Without keys: pieces are indistinguishable in result
result = pd.concat([s1, s1, s3], keys=['one','two','three'])
print(result)
# one  a  0
#    b  1
# two  a  0
#    b  1
# three f  5
#    g  6

# Unstack to see as DataFrame
print(result.unstack())
```

► **Concatenating DataFrames with ignore_index**

Python Code

```
df1 = pd.DataFrame(np.random.randn(3,4), columns=['a','b','c','d'])
df2 = pd.DataFrame(np.random.randn(2,3), columns=['b','d','a'])

# ignore_index: reset row index to 0,1,2,3,4
result = pd.concat([df1, df2], ignore_index=True)
print(result)
# Column c has NaN for df2 rows (df2 had no column c)
```

concat() Parameter	Description
objs	List or dict of pandas objects to concatenate (required)
axis	0 = stack rows (default). 1 = stack columns side-by-side.
join	'outer' (union, default) or 'inner' (intersection) on other axes
join_axes	Specific index to use for other axes instead of union/intersection
keys	Create hierarchical index to identify each piece in the result
names	Names for hierarchical index levels created by keys

concat() Parameter	Description
ignore_index	If True, reset integer index 0,1,2,... instead of preserving original
verify_integrity	If True, raise error on duplicate indexes in result

Combining Data with Overlap — combine_first()

combine_first() fills NaN values in one Series/DataFrame with corresponding values from another. Think of it as "patching" missing data:

Python Code

```
import numpy as np

a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
              index=['f','e','d','c','b','a'])
b = pd.Series(np.arange(len(a), dtype=np.float64),
              index=['f','e','d','c','b','a'])
b[-1] = np.nan

# combine_first: fill NaN in b with values from a
result = b[:-2].combine_first(a[2:])
print(result)
# a  NaN  (both NaN, so stays NaN)
# b  4.5  (from a, since b slice has no b)
# c  3.0  (from b)
# d  2.0  (from b)
# e  1.0  (from b)
# f  0.0  (from b)

# With DataFrames — fills column-by-column
df1 = pd.DataFrame({'a':[1,np.nan,5,np.nan], 'b':[np.nan,2,np.nan,6],
                    'c':range(2,18,4)})
df2 = pd.DataFrame({'a':[5,4,np.nan,3,7], 'b':[np.nan,3,4,6,8]})
print(df1.combine_first(df2))
```

```
   a  b  c
0  1.0 NaN 2
1  4.0 2.0 6
2  5.0 4.0 10
3  3.0 6.0 14
4  7.0 8.0 NaN
```

2.2 Reshaping and Pivoting

Reshaping rearranges data between different structural formats without changing its values. The two core operations are stack/unstack (for hierarchical indexes) and pivot/melt (for wide/long format conversion).

Reshaping with Hierarchical Indexing — stack() and unstack()

These two methods rotate between rows and columns in a DataFrame:

- stack(): "rotates" columns into rows → returns a Series with MultiIndex
- unstack(): "rotates" rows into columns → returns a DataFrame

Python Code

```
import pandas as pd, numpy as np

data = pd.DataFrame(np.arange(6).reshape((2,3)),
                    index=pd.Index(['Ohio','Colorado'], name='state'),
                    columns=pd.Index(['one','two','three'], name='number'))

print(data)
# number one two three
# state
# Ohio    0  1  2
# Colorado 3  4  5

# STACK: columns → rows (produces MultiIndex Series)
stacked = data.stack()
print(stacked)
# state  number
# Ohio   one    0
#        two    1
#        three  2
# Colorado one    3
#         two    4
#         three  5

# UNSTACK: reverse — rows → columns
print(stacked.unstack())
```

```
# Gives back the original DataFrame!
```

► Unstacking a Specific Level

Python Code

```
# By default unstack() uses the innermost level
# Pass level number or name to change:

# Unstack level 0 (the outer "state" level)
print(stacked.unstack(0))

# Unstack by name
print(stacked.unstack('state'))

# Output for both:
# state    Ohio  Colorado
# number
# one      0      3
# two      1      4
# three    2      5
```

► Unstacking with Missing Data

Python Code

```
s1 = pd.Series([0,1,2,3], index=['a','b','c','d'])
s2 = pd.Series([4,5,6], index=['c','d','e'])
data2 = pd.concat([s1, s2], keys=['one','two'])

print(data2.unstack())
#    a  b  c  d  e
# one 0.0 1.0 2 3 NaN
# two NaN NaN 4 5 6.0

# Stack filters out NaN by default (easily invertible):
print(data2.unstack().stack())

# Stack with dropna=False to keep NaN:
print(data2.unstack().stack(dropna=False))
```

Pivoting "Long" to "Wide" Format

A "long" table stores multiple measurements as separate rows — one row per (date, variable) combination. This is common in database storage but inconvenient for analysis. `pivot()` converts it to a "wide" table with one column per variable.

Python Code

```
# Long format (how data is often stored in databases)
ldata = pd.DataFrame({
    'date': ['1959-03-31']*3 + ['1959-06-30']*3,
    'item': ['realgdp','infl','unemp']*2,
    'value': [2710.35, 0.00, 5.80, 2778.80, 2.34, 5.10]})

print(ldata)
# date      item  value
# 1959-03-31 realgdp 2710.35
# 1959-03-31 infl    0.00
# 1959-03-31 unemp    5.80
# ...

# PIVOT: date=index, item=columns, value=cell data
pivoted = ldata.pivot('date', 'item', 'value')
print(pivoted)
```

```
item      infl realgdp unemp
date
1959-03-31 0.00 2710.35 5.8
1959-06-30 2.34 2778.80 5.1
```

► Pivoting with Multiple Value Columns

Python Code

```
# Add a second value column
ldata['value2'] = np.random.randn(len(ldata))

# Omit the value argument → MultiIndex columns
pivoted = ldata.pivot('date', 'item')
print(pivoted[:3])

# Result has hierarchical columns:
#      value      value2
# item  infl realgdp  infl realgdp ...
```



```
# Access specific value column:
print(pivoted['value'])

# NOTE: pivot is a shortcut for:
unstacked = ldata.set_index(['date','item']).unstack('item')
```

Pivoting "Wide" to "Long" Format — `pd.melt()`

Python Code

```
df = pd.DataFrame({'key': ['foo','bar','baz'],
                    'A': [1,2,3], 'B': [4,5,6], 'C': [7,8,9]})

# Melt: keep key as identifier, convert A/B/C to rows
melted = pd.melt(df, id_vars=['key'])
print(melted)

# key variable value
# foo      A      1
# bar      A      2
# baz      A      3
# foo      B      4
# ...

# Specify which value columns to melt
melted2 = pd.melt(df, id_vars=['key'], value_vars=['A','B'])
```

2.3 Data Transformation

Data transformation covers a wide range of operations: removing duplicates, replacing sentinel values, mapping categories, renaming indexes, discretizing continuous data into bins, filtering outliers, sampling, and computing indicator (dummy) variables.

A — Removing Duplicates

Python Code

```
data = pd.DataFrame({'k1': ['one']*3 + ['two']*4,
                     'k2': [1,1,2,3,3,4,4]})

# Check which rows are duplicates
print(data.duplicated())
# 0 False (first occurrence)
# 1 True  (duplicate of row 0)
# 2 False (k2=2 is new)
```

```
# 3 False
# 4 True (duplicate of row 3)
# 5 False
# 6 True

# Count total duplicates
print(data.duplicated().sum()) # 3

# Remove duplicates (keep FIRST by default)
clean = data.drop_duplicates()

# Keep LAST occurrence instead
clean_last = data.drop_duplicates(keep='last')

# Drop duplicates based on specific column only
clean_k1 = data.drop_duplicates(['k1'])

# Based on multiple columns
clean_both = data.drop_duplicates(['k1','k2'])
```

B — Replacing Values

The `replace()` method is more general than `fillna()` — it can replace any value, not just NaN:

Python Code

```
data = pd.Series([1., -999., 2., -999., -1000., 3.])

# Replace single value with NaN
print(data.replace(-999, np.nan))

# Replace multiple values — all get same replacement
print(data.replace([-999, -1000], np.nan))

# Replace multiple values — different replacement for each
print(data.replace([-999, -1000], [np.nan, 0]))

# Using a dict mapping
print(data.replace({-999: np.nan, -1000: 0}))

# Works on DataFrame too
df = pd.DataFrame({'a': [0,1,2,3], 'b': [5,-1,2,-1]})
print(df.replace(-1, np.nan))
```

```
0  1.0
1  NaN
2  2.0
3  NaN
4  0.0
5  3.0
```

C — Transforming with `map()` and `apply()`

`map()` applies a function or dict lookup element-wise to a Series. It is ideal for encoding categorical variables:

Python Code

```
data = pd.DataFrame({
    'food': ['bacon','pulled pork','bacon','Pastrami',
            'corned beef','Bacon','pastrami','honey ham','nova lox'],
    'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

# Dict mapping food → animal
meat_to_animal = {
    'bacon': 'pig', 'pulled pork': 'pig',
    'pastrami': 'cow', 'corned beef': 'cow',
    'honey ham': 'pig', 'nova lox': 'salmon'
}

# Problem: some foods are capitalized — normalize first
data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
print(data)

# Or use a lambda for one-liner
data['animal2'] = data['food'].map(lambda x: meat_to_animal[x.lower()])
```

```
   food ounces animal
0  bacon   4.0   pig
3 Pastrami   6.0   cow
4 corned beef  7.5   cow
8  nova lox   6.0 salmon
```

D — Renaming Axis Indexes

Use `.map()` on an index or the `rename()` method to relabel row/column indexes:

Python Code

```
data = pd.DataFrame(np.arange(12).reshape((3,4)),
                    index=['Ohio','Colorado','New York'],
                    columns=['one','two','three','four'])

# Method 1: Assign directly (modifies in place)
data.index = data.index.map(str.upper)
print(data.index) # [OHIO, COLORADO, NEW YORK]

# Method 2: rename() — returns new object
renamed = data.rename(index=str.title, columns=str.upper)
print(renamed)

# Method 3: rename() with dict — partial relabeling
partial = data.rename(index={'OHIO': 'INDIANA'},
                      columns={'three': 'peekaboo'})
print(partial)

# In-place with inplace=True
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

E — Discretization and Binning

Discretization converts continuous numerical data into categorical buckets. This is useful for age groups, income ranges, score grades, etc.

► pd.cut() — Fixed Bin Edges**Python Code**

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

# Define bin edges
bins = [18, 25, 35, 60, 100]

# Cut into bins
cats = pd.cut(ages, bins)
print(cats)
# Categorical: [(18, 25], (18, 25], (18, 25], (25, 35], ...]

# Inspect labels and levels
print(cats.codes) # Integer labels: [0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1]
print(cats.categories)
```

```
# Count per bin
print(pd.value_counts(cats))
# (18, 25]    5
# (25, 35]    3
# (35, 60]    3
# (60, 100]   1

# Custom labels
groups = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
labeled = pd.cut(ages, bins, labels=groups)
print(labeled)

# Right=False: intervals include left edge [18,25) instead of (18,25]
print(pd.cut(ages, [18,26,36,61,100], right=False))

# Auto bins: pass integer instead of edges
data = np.random.rand(20)
print(pd.cut(data, 4, precision=2)) # 4 equal-width bins
```

► **pd.qcut()** — Quantile-Based Bins (Equal-Size)

Unlike `cut()`, `qcut()` creates bins with equal numbers of data points (by using quantiles):

Python Code

```
data = np.random.randn(1000)

# Cut into quartiles (4 equal groups)
cats = pd.qcut(data, 4)
print(pd.value_counts(cats))
# Each bin has exactly 250 values

# Custom quantiles
custom = pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.0])
# Creates bins at 10th, 50th, 90th percentiles
```

pd.cut() Parameter	Description
x	Input 1D array-like of values to bin
bins	Int (number of equal-width bins) or sequence of bin edges
right	True (default): bins are (left, right]. False: [left, right)
labels	Array of labels for bins. Length = len(bins)-1
precision	Precision for auto-computed bin boundaries

pd.cut() Parameter	Description
include_lowest	If True, first interval includes left edge

F — Detecting and Filtering Outliers

Python Code

```
np.random.seed(12345)
data = pd.DataFrame(np.random.randn(1000, 4))

# View statistics
print(data.describe())
# min can be -3.4, max can be 3.9...

# Find values in column 3 exceeding 3 in magnitude
col = data[3]
print(col[np.abs(col) > 3])
# 97    3.927528
# 305   -3.399312
# 400   -3.745356

# Select ALL rows where ANY column has abs value > 3
print(data[(np.abs(data) > 3).any(axis=1)])

# Cap outliers at ±3
data[np.abs(data) > 3] = np.sign(data) * 3

# Verify: min=-3, max=3
print(data.describe().loc[["min", "max"]])
# np.sign() returns +1 or -1 based on sign of each value
```

G — Permutation and Random Sampling

Python Code

```
df = pd.DataFrame(np.arange(5*4).reshape(5,4))

# Create a random permutation of row indices
sampler = np.random.permutation(5)
print(sampler) # e.g. [3, 1, 4, 0, 2]

# Reorder rows using .take()
print(df.take(sampler))
```

```
# Random sample of 3 rows (without replacement)
subset = df.take(np.random.permutation(len(df))[:3])
print(subset)

# Bootstrap sample (with replacement)
bag = np.array([5, 7, -1, 6, 4])
sampler2 = np.random.randint(0, len(bag), size=10)
draws = bag.take(sampler2)
print(draws) # e.g. [4, 4, -1, -1, 5, 6, 5, 4, 7, -1]
```

H — Computing Indicator/Dummy Variables

Categorical variables must be converted to numeric (0/1) binary indicator columns for use in most statistical models and machine learning algorithms. `pd.get_dummies()` does this automatically:

Python Code

```
df = pd.DataFrame({'key': ['b','b','a','c','a','b'], 'data1': range(6)})

# Generate dummy variables for the key column
dummies = pd.get_dummies(df['key'])
print(dummies)
# a b c
# 0 0 1 0
# 1 0 1 0
# 2 1 0 0
# 3 0 0 1

# Add prefix to avoid column name conflicts
dummies = pd.get_dummies(df['key'], prefix='key')

# Combine with original DataFrame
df_with_dummies = df[['data1']].join(dummies)
print(df_with_dummies)
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1

► Multi-label Dummy Variables (e.g., Movie Genres)**Python Code**

```
# Each movie can have multiple genres separated by |
movies = pd.read_table('movies.dat', sep='::', header=None,
                      names=['movie_id', 'title', 'genres'])

# Extract all unique genres
genre_iter = (set(x.split('|')) for x in movies.genres)
genres = sorted(set.union(*genre_iter))

# Build zero DataFrame
dummies = pd.DataFrame(np.zeros((len(movies), len(genres))), columns=genres)

# Fill 1 for each genre a movie belongs to
for i, gen in enumerate(movies.genres):
    dummies.ix[i, gen.split('|')] = 1

# Join back with movies
movies_windic = movies.join(dummies.add_prefix('Genre_'))

# Combining get_dummies with cut (useful in statistics)
values = np.random.rand(10)
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
print(pd.get_dummies(pd.cut(values, bins)))
```

2.4 String Manipulation

Python is a beloved language for text processing. String manipulation in data analysis involves: cleaning messy values, extracting patterns, parsing emails/URLs/dates, standardizing case and whitespace, and encoding categorical text.

Three Levels of String Tools

- Python built-in string methods — simple and fast for basic operations
- Python re module — powerful regular expressions for pattern matching and extraction
- pandas .str accessor — vectorized string methods that handle NaN automatically

Every Python string object has many useful methods:

Python Code

```
val = 'a,b, guido'

# SPLIT: break into list
print(val.split(',')) # ['a', 'b', ' guido']

# STRIP + SPLIT: split and clean whitespace
pieces = [x.strip() for x in val.split(',')]
print(pieces)        # ['a', 'b', 'guido']

# JOIN: concatenate with delimiter
print('::'.join(pieces)) # 'a::b::guido'

# IN: check membership
print('guido' in val) # True

# INDEX: find position (raises error if not found)
print(val.index(',')) # 1

# FIND: find position (returns -1 if not found)
print(val.find(',')) # -1 ← safer than index

# COUNT: count occurrences
print(val.count(',')) # 2

# REPLACE: substitute occurrences
print(val.replace(',', '::')) # 'a::b:: guido'
print(val.replace(',', '')) # 'ab guido'

# UPPER / LOWER
print('Hello World'.upper()) # HELLO WORLD
print('Hello World'.lower()) # hello world

# STARTSWITH / ENDSWITH
print('Hello'.startswith('He')) # True
print('Hello'.endswith('lo')) # True
```

Method	Returns	Description
split(sep)	list	Split string by sep; returns list of substrings
strip() / lstrip() / rstrip()	str	Remove whitespace from both/left/right ends
join(iterable)	str	Join list of strings using this string as separator
lower() / upper()	str	Convert all chars to lower/uppercase
replace(old, new)	str	Replace all occurrences of old with new
count(sub)	int	Count non-overlapping occurrences of substring
index(sub)	int	First position of substring; raises ValueError if absent
find(sub)	int	First position of substring; returns -1 if absent
rfind(sub)	int	Last position of substring; -1 if absent
startswith(prefix)	bool	True if string begins with prefix
endswith(suffix)	bool	True if string ends with suffix
ljust(width) / rjust(width)	str	Left/right justify with space padding
in operator	bool	Check membership: 'sub' in string

B — Regular Expressions (re Module)

Regular expressions provide a highly flexible way to search, match, and transform string patterns. The re module provides this functionality in Python.



SYNTAX

Import re before use: `import re`. Create compiled patterns with `re.compile()` for performance when applying the same pattern many times.

► Basic Pattern Matching — `split`, `findall`, `search`, `match`

Python Code

```
import re

# SPLIT: break string on whitespace
text = 'foo bar\t baz \tqux'
print(re.split('\s+', text))
# ['foo', 'bar', 'baz', 'qux']

# Compile regex for reuse
```

```
regex = re.compile('\s+')
print(regex.split(text))
# ['foo', 'bar', 'baz', 'qux']

# FINDALL: find all non-overlapping matches
print(regex.findall(text))
# [' ', '\t', '\t'] ← the whitespace groups themselves
```

► **Email Address Extraction — A Practical Example**

Python Code

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""

pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
regex = re.compile(pattern, flags=re.IGNORECASE)

# FINDALL: extract all email addresses
emails = regex.findall(text)
print(emails)
# ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']

# SEARCH: find first match (returns match object)
m = regex.search(text)
print(m.group())    # dave@google.com
print(m.start())    # 5
print(m.end())      # 19

# MATCH: only matches at start of string
print(regex.match(text)) # None (text starts with Dave, not email)

# SUB: replace matches
censored = regex.sub("REDACTED", text)
print(censored)
# Dave REDACTED
# Steve REDACTED ...
```

```
# Use parentheses () to define capture groups
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})'
regex = re.compile(pattern, flags=re.IGNORECASE)

# groups() returns tuple of captured groups
m = regex.match('wesm@bright.net')
print(m.groups())
# ('wesm', 'bright', 'net')

# findall with groups returns list of tuples
print(regex.findall(text))
# [('dave', 'google', 'com'), ('steve', 'gmail', 'com'), ...]

# sub with back-references \1, \2, \3
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
# Dave Username: dave, Domain: google, Suffix: com
# Steve Username: steve, Domain: gmail, Suffix: com ...

# Named groups using (?P<name>...)
regex2 = re.compile(r'(?P<username>[A-Z0-9._%+-]+)@(?P<domain>[A-Z0-9.-]+\.(?P<suffix>[A-Z]{2,4}))', flags=re.IGNORECASE|re.VERBOSE)

m2 = regex2.match('wesm@bright.net')
print(m2.groupdict())
# {'username': 'wesm', 'domain': 'bright', 'suffix': 'net'}
```

Pattern	Meaning	Example Match
.	Any single character (except newline)	Any char: 'a', '3', '@'
\d	Digit: [0-9]	'5', '0', '9'
\D	Non-digit: [^0-9]	'a', ',', '@'
\w	Word char: [a-zA-Z0-9_]	'hello_1'
\W	Non-word character	',';', '!'
\s	Whitespace: space, tab, newline	',';'\t'

Pattern	Meaning	Example Match
\S	Non-whitespace	'hello'
+	One or more of preceding	'a', 'aaa'
*	Zero or more of preceding	'', 'aaa'
?	Zero or one (optional)	'' or 'a'
{n}	Exactly n repetitions	'.com' for [a-z]{3}
{n,m}	Between n and m repetitions	'.co' to '.info'
^	Start of string/line	'^Hello' matches 'Hello world'
\$	End of string/line	'world\$' matches 'Hello world'
[abc]	Any char in set: a or b or c	'a', 'b', 'c'
[^abc]	Any char NOT in set	'd', 'e', '1'
(group)	Capture group	m.group(1), m.groups()
(?P<name>)	Named capture group	m.groupdict()
a b	a OR b (alternation)	'cat' or 'dog'

re Function/Method	Description
re.compile(pattern, flags)	Compile pattern into reusable regex object
regex.findall(string)	List of all non-overlapping matches
regex.finditer(string)	Iterator over matches (memory-efficient for large text)
regex.search(string)	First match anywhere in string; returns match object or None
regex.match(string)	Match only at START of string; returns match object or None
regex.split(string)	Split string at each match; like str.split but with patterns
regex.sub(repl, string)	Replace all matches with repl string; can use back-refs \1 \2
regex.subn(repl, string)	Like sub() but also returns count of substitutions made
m.group()	Return the full match or group n from match object m
m.groups()	Return tuple of all captured groups from match object
m.groupdict()	Return dict of named groups from match object
m.start(), m.end()	Start and end positions of match in original string

When a pandas Series contains strings, applying string/regex operations using `.map()` will fail on NaN values. The `.str` accessor solves this — it applies string operations element-wise and automatically propagates NaN:

Python Code

```
data = pd.Series({'Dave': 'dave@google.com',
                  'Steve': 'steve@gmail.com',
                  'Rob': 'rob@gmail.com',
                  'Wes': np.nan})

# Check for NaN
print(data.isnull())
# Dave False Steve False Rob False Wes True

# str.contains: check substring (NaN is handled gracefully)
print(data.str.contains('gmail'))
# Dave False Rob True Steve True Wes NaN

# Regex findall
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})'
print(data.str.findall(pattern, flags=re.IGNORECASE))
# Dave [('dave','google','com')]
# Rob  [('rob','gmail','com')]
# Wes  NaN

# String slicing
print(data.str[:5])
# Dave dave@ Rob rob@g Steve steve Wes NaN

# Vectorized match + extract parts
matches = data.str.match(pattern, flags=re.IGNORECASE)
print(matches.str.get(1)) # Get domain part
# Dave google Rob gmail Steve gmail Wes NaN
```

► Complete Vectorized String Methods Reference

Method	Description
str.contains(pattern)	Boolean: does each string contain the pattern?
str.count(pattern)	Count occurrences of pattern in each string
str.startswith(pattern)	Boolean: does each string start with pattern?
str.endswith(pattern)	Boolean: does each string end with pattern?

Method	Description
str.findall(pattern)	List of all matches per element (handles NaN)
str.match(pattern)	Match pattern at start; return matched groups as list
str.get(i)	Get i-th element from each (for list-like elements)
str.join(sep)	Join list elements in each row with separator
str.len()	Length of each string element
str.lower() / str.upper()	Convert to lowercase / uppercase
str.title()	Title-case each string (first letter of each word capitalized)
str.replace(pat, repl)	Replace occurrences of pattern with replacement
str.split(pattern)	Split each string by delimiter/pattern → list
str.strip() / lstrip() / rstrip()	Remove whitespace from ends
str.pad(width, side)	Pad strings to given width with spaces (left, right, both)
str.center(width)	Center string with space padding
str.repeat(n)	Repeat each string n times
str.slice(start, stop)	Slice each string — equivalent to str[start:stop]
str.cat(sep)	Concatenate strings element-wise with delimiter

Chapter 1 Summary — Data Loading & Storage

Task	Primary Function	Key Parameters
Read CSV	pd.read_csv(file)	sep, header, index_col, names, na_values, nrows, chunksize
Read tab-delimited	pd.read_table(file)	sep='\t', header, skiprows, encoding
Read large file in chunks	pd.read_csv(chunksize=N)	Iterate: for piece in chunker: ...
Write CSV	df.to_csv(file)	sep, index, header, na_rep, columns
Pickle (save)	df.to_pickle(file)	—
Pickle (load)	pd.read_pickle(file)	—
HDF5 (write)	df.to_hdf(file, 'key')	format='fixed' or 'table'
HDF5 (read)	pd.read_hdf(file, 'key')	where=[filter expression]
Excel (read)	pd.read_excel(file)	sheet_name, index_col
Excel (write)	df.to_excel(file)	sheet_name, index
SQL (read)	pd.read_sql('SELECT...', con)	index_col, parse_dates
SQL (write)	df.to_sql('table', con)	if_exists, index
JSON (parse)	json.loads(string)	→ dict/list; wrap in DataFrame()
HTML scrape	lxml.html.parse(url)	doc.findall('.//table')
Web API	requests.get(url)	json.loads(resp.text)

Operation	Function	Key Parameters / Notes
Join DataFrames	pd.merge(df1, df2)	on, left_on, right_on, how='inner/left/right/outer'
Join on index	df1.join(df2)	how='outer', on=col_name
Stack vertically	pd.concat([df1,df2])	axis=0, keys, ignore_index, join
Stack horizontally	pd.concat([df1,df2], axis=1)	Creates side-by-side DataFrame
Fill NaN from other	df1.combine_first(df2)	Column-by-column patching
Cols → rows	df.stack()	level=, dropna=True
Rows → cols	series.unstack()	level=, fill_value=
Long → wide	df.pivot(idx, col, val)	index, columns, values args
Wide → long	pd.melt(df)	id_vars, value_vars
Remove duplicates	df.drop_duplicates()	subset=, keep='first'/'last'
Replace values	df.replace(old, new)	List or dict mapping
Map categories	series.map(dict_or_fn)	Use str.lower() to normalize first
Rename labels	df.rename()	index=dict, columns=dict, inplace=True
Bin continuous	pd.cut(x, bins)	bins, labels, right, precision
Equal-size bins	pd.qcut(x, q)	q=int or [quantile list]
Filter outliers	df[np.abs(df) > 3]	Use .any(axis=1) for row selection
Random sample	df.take(np.random.permutation(n)[:k])	Without replacement
Dummy variables	pd.get_dummies(series)	prefix=, drop_first=True

Category	Key Methods/Functions	When to Use
Python built-in	split, strip, join, replace, upper, lower, find, count, startswith, endswith	Simple operations on individual strings
re module	re.compile, findall, search, match, sub, split, groups()	Pattern matching, extraction, complex substitution
pandas .str	str.contains, str.findall, str.replace, str.split, str.lower, str.get, str.len	Vectorized operations on entire Series with NaN safety