ROHINI COLLEGE OF ENGINEERING & TECHNOLOGY
(AUTONOMOUS)
Approved by AICTE & Affiliated to Anna University | All Eligible Courses Accredited by NBA | Accredited by NAAC with A+ Grade

DEPARTMENT OF BIOMEDICAL ENGINEERING

24AI671- FUNDAMENTALS OF DATA SCIENCE

UNIT II

DATA COLLECTION, CLEANING AND PREPROCESSING

Syllabus topic
Data Acquisition Methods
Data Storage: CSV, SQL, NoSQL
Missing Data Handling
Outlier Detection
Noise Reduction
Feature Engineering
Data Transformation and Scaling
Data Wrangling Tools – Python, Pandas, NumPy
Unit summary, revision table and question bank

8. DATA WRANGLING TOOLS – PYTHON, PANDAS, NUMPY

Data wrangling (data munging) is the process of **transforming raw data into a clean, structured and usable form** for analysis. It covers all the activities of this unit – acquiring, loading, cleaning, handling missing values and outliers, reshaping, merging, feature creation and scaling. Surveys report that data scientists spend **50–80% of their time** on wrangling rather than on modelling.

8.1 Steps in Data Wrangling

Step	Activity	Example
1. Discover	Understand the data – size, columns, types, quality	head(), info(), describe()

Step	Activity	Example
2. Structure	Organise into rows and columns; reshape	melt(), pivot(), JSON → table
3. Clean	Fix missing values, duplicates, outliers, errors	fillna(), drop_duplicates()
4. Enrich	Add data from other sources; create features	merge(), new columns
5. Validate	Check ranges, types and rules	assert statements, value counts
6. Publish	Save the final data for modelling	to_csv(), to_sql()

8.2 Python – The Language of Data Science

Python is a high-level, interpreted, general-purpose language and is the most popular language for data science because of its **simple syntax, free and open source licence, huge community and a very rich set of libraries**. Code is usually written in a **Jupyter Notebook** or Google Colab, which combines code, output, graphs and notes in one document.

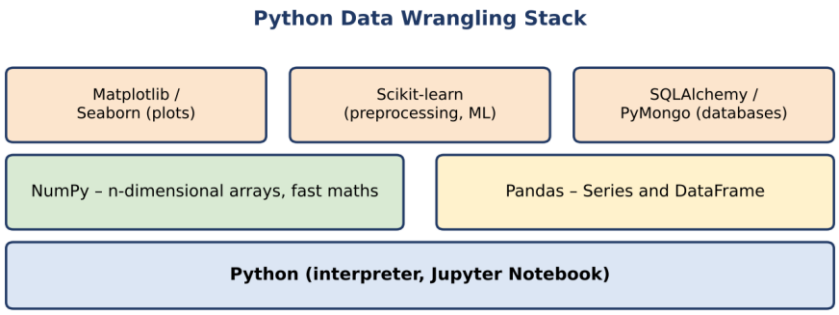


Fig 8.1 Python data wrangling stack

Library	Purpose
NumPy	Fast numerical computing with n-dimensional arrays
Pandas	Tabular data (Series and DataFrame): cleaning, merging, grouping
Matplotlib / Seaborn	Data visualisation – line, bar, histogram, box plots, heat-maps
SciPy	Scientific computing – statistics, signal filters, optimisation
Scikit-learn	Pre-processing, feature selection, ML algorithms, evaluation
Requests / BeautifulSoup	Web access and scraping
SQLAlchemy / sqlite3 / PyMongo	Connecting to SQL and NoSQL databases

Code

```
# Installing and importing (run in a terminal / notebook cell)
# pip install numpy pandas matplotlib scikit-learn

import numpy as np          # np  is the standard alias
import pandas as pd         # pd  is the standard alias
import matplotlib.pyplot as plt
```

8.3 NumPy – Numerical Python

NumPy provides the **ndarray** – a fast, fixed-type, n-dimensional array stored in a continuous block of memory. NumPy arrays are **much faster and use less memory than Python lists** because the operations are done in compiled C code (**vectorisation**) rather than in Python loops. NumPy is the base on which Pandas, SciPy and Scikit-learn are built.

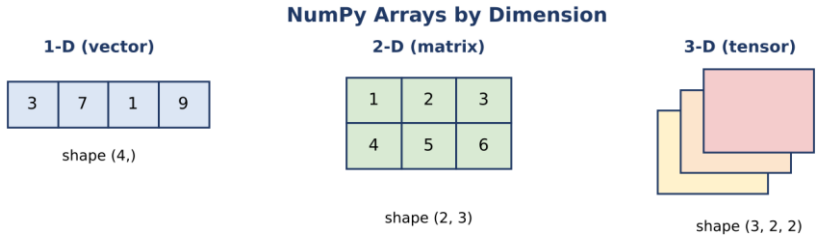


Fig 8.2 NumPy arrays of one, two and three dimensions

NumPy array vs Python list. A Python list can hold mixed types (numbers, strings, other lists) and each element is a full Python object, so it is flexible but slow and memory-hungry. A NumPy array holds **one data type** in a continuous block of memory, so mathematical operations are done by fast compiled code. For example, to double every number a list needs a loop or comprehension (`[v*2 for v in lst]`), whereas for a NumPy array the single expression `arr * 2` does it – the code is shorter and, for large data, many times faster. Also note that `[1,2,3] * 2` on a list repeats the list ([1,2,3,1,2,3]) while `np.array([1,2,3]) * 2` multiplies each element ([2,4,6]).

Understanding `axis`. In a 2-D array, **axis = 0 runs down the rows** (so an operation works column by column) and **axis = 1 runs across the columns** (so it works row by row). That is why `m.mean(axis=0)` gives one mean for each column and `m.sum(axis=1)` gives one total for each row. The same idea is used in Pandas (`df.sum(axis=1)`, `df.dropna(axis=1)`).

Creating arrays and important attributes

Code

```
import numpy as np

a = np.array([1, 2, 3])                  # 1-D from a list
m = np.array([[1, 2, 3], [4, 5, 6]])        # 2-D (2 rows, 3 columns)

print(m.shape, m.ndim, m.size, m.dtype)      # (2, 3) 2 6 int64

np.arange(0, 10, 3)       # [0 3 6 9]          start, stop, step
```

```
np.linspace(0, 1, 5)      # [0.  0.25 0.5 0.75 1.]   5 equally spaced numbers
np.zeros((2, 2))          # 2x2 array of 0.0
np.ones((2, 3))           # 2x3 array of 1.0
np.eye(2)                 # 2x2 identity matrix
np.random.rand(3)         # 3 random numbers between 0 and 1
```

Indexing, slicing and boolean masking

Code

```
m = np.array([[1, 2, 3], [4, 5, 6]])

print(m[1, 2])       # 6          element at row 1, column 2
print(m[:, 1])       # [2 5]      all rows, column 1
print(m[1, :2])      # [4 5]      row 1, first two columns
print(m[m > 2])      # [3 4 5 6]  boolean mask: elements greater than 2
```

Vectorised operations and broadcasting

Operations are applied to **every element at once**, without loops. **Broadcasting** lets NumPy combine arrays of different shapes by stretching the smaller array automatically, provided the shapes are compatible.

Code

```
x = np.array([1, 2, 3])
print(x * 2 + 1)            # [3 5 7]     element-wise, no loop

print(m + np.array([10, 20, 30]))   # broadcasting: row added to every row of m
```

Output

```
[3 5 7]
[[11 22 33]
 [14 25 36]]
```

Aggregation, reshaping and missing values

Code

```
print(m.mean(axis=0))      # [2.5 3.5 4.5]   column-wise mean (axis=0)
print(m.sum(axis=1))       # [ 6 15]         row-wise sum (axis=1)
print(m.reshape(3, 2))     # change shape without changing data
print(m.T.shape)           # (3, 2)          transpose

n = np.array([1, np.nan, 3])
print(np.isnan(n))         # [False  True False]
print(n.mean())            # nan             NaN spreads through the calculation
print(np.nanmean(n))       # 2.0             ignores NaN
```

Task	NumPy function
Statistics	np.mean, np.median, np.std, np.var, np.min, np.max, np.percentile, np.corrcoef

Task	NumPy function
Maths	np.sqrt, np.exp, np.log, np.abs, np.power, np.dot (dot product), np.matmul
Shape changes	reshape, flatten, transpose, np.concatenate, np.vstack, np.hstack, np.split
Searching and sorting	np.where, np.argmax, np.argmin, np.sort, np.unique
Missing values	np.isnan, np.nanmean, np.nansum, np.nan_to_num
Linear algebra	np.linalg.inv, np.linalg.det, np.linalg.eig, np.linalg.solve

Why NumPy is faster than a Python list

- Elements have the **same data type** and are stored **contiguously** in memory.
- Operations run in **optimised C / SIMD code**; there is no Python loop overhead.
- Uses **less memory** because it does not store a full Python object for each number.

8.4 Pandas – Data Analysis Library

Pandas (Python Data Analysis Library) is built on NumPy and provides two main data structures for labelled data:

- **Series** – a one-dimensional labelled array (a single column).
- **DataFrame** – a two-dimensional table with labelled rows (**index**) and columns, where each column is a Series and may have a different data type.

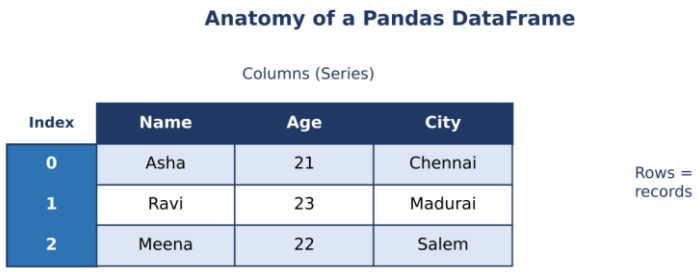


Fig 8.3 Structure of a Pandas DataFrame

Creating and inspecting a DataFrame

Code

```
import pandas as pd

s = pd.Series([10, 20, 30], index=["a", "b", "c"])        # Series

emp = pd.DataFrame({
    "Name":   ["Asha", "Ravi", "Meena", "Kiran", "Divya", "Arun"],
    "Dept":   ["IT", "HR", "IT", "HR", "IT", "Sales"],
    "Salary": [50000, 42000, 58000, 45000, 62000, 39000],
    "Exp":    [3, 2, 5, 4, 7, 1]})
```

```
emp.head()       # first 5 rows          emp.tail()     # last 5 rows
emp.shape        # (6, 4)               emp.columns    # column names
emp.dtypes       # data type of each col   emp.info()     # summary of columns
print(emp.describe().round(1))            # statistics of numeric columns
```

Output

```
       Salary  Exp
count     6.0  6.0
mean   49333.3  3.7
std     9114.1  2.2
min    39000.0  1.0
25%    42750.0  2.2
50%    47500.0  3.5
75%    56000.0  4.8
max    62000.0  7.0
```

Selecting, filtering and sorting

Code

```
emp["Name"]                    # one column  -> Series
emp[["Name", "Salary"]]           # several columns -> DataFrame
# label-based: rows 0 to 2 (inclusive), 2 columns
emp.loc[0:2, ["Name", "Dept"]]
emp.iloc[0:2, 0:2]               # position-based: rows 0-1, columns 0-1

# filtering with conditions
print(emp[(emp.Dept == "IT") & (emp.Salary > 55000)])
print(emp.sort_values("Salary", ascending=False).head(3))   # top 3 salaries
```

Output

```
   Name Dept  Salary  Exp
2  Meena   IT   58000    5
4  Divya   IT   62000    7

   Name Dept  Salary  Exp
4  Divya   IT   62000    7
2  Meena   IT   58000    5
0   Asha   IT   50000    3
```

Use `&` (and), `|` (or) and `~` (not) with **brackets around each condition** when combining filters.

Point	loc	iloc
Selection by	Row / column **labels** (names)	Integer **positions** (0, 1, 2 …)
Slice end	**Inclusive** – emp.loc[0:2] gives rows 0, 1 and 2	**Exclusive** – emp.iloc[0:2] gives rows 0 and 1

Point	loc	iloc
Example	`emp.loc[emp.Dept == "IT", "Name"]`	`emp.iloc[0, 0]`
Conditions	Accepts boolean masks	Positions only (mask must be converted)

For the employee table, `emp.loc[emp.Dept == "IT", "Name"]` returns Asha, Meena and Divya, and `emp.iloc[0, 0]` returns the first cell, ‘Asha’.

Grouping, aggregation and pivot tables

Code

```
print(emp.groupby("Dept")["Salary"].mean())
print(emp.groupby("Dept").agg(avg=("Salary", "mean"), n=("Name", "count")))
print(emp.pivot_table(values="Salary", index="Dept", aggfunc="sum"))
```

Output

```
Dept
HR       43500.000000
IT       56666.666667
Sales    39000.000000
Name: Salary, dtype: float64

              avg  n
Dept
HR     43500.000000  2
IT     56666.666667  3
Sales  39000.000000  1

       Salary
Dept
HR      87000
IT     170000
Sales   39000
```

Split–apply–combine. The `groupby()` function works in three steps: it **splits** the table into groups according to a key column (for example Dept), **applies** a function such as mean, sum or count to every group, and **combines** the results into a new table. In the output above, the IT group contains Asha, Meena and Divya, so its average salary is (50,000 + 58,000 + 62,000) / 3 = ₹ 56,667. `pivot_table` does the same thing but arranges the groups in a matrix of rows and columns, similar to a pivot table in Excel.

Combining datasets – merge, join, concat

Code

```
dept = pd.DataFrame({"Dept": ["IT", "HR", "Sales"],
                     "Head": ["Mr. Raj", "Ms. Uma", "Mr. Sam"]})

both = emp.merge(dept, on="Dept", how="left")   # like SQL JOIN
print(both.head(3))
more = pd.concat([emp, emp.tail(1)], ignore_index=True)   # stack rows
```

Output

```
    Name Dept  Salary  Exp     Head
0   Asha   IT   50000    3  Mr. Raj
1   Ravi   HR   42000    2  Ms. Uma
2  Meena   IT   58000    5  Mr. Raj
```

merge type (how=)	Result
inner	Only rows whose key is present in both tables
left	All rows of the left table + matching rows of the right
right	All rows of the right table + matching rows of the left
outer	All rows of both tables; NaN where no match

Cleaning and transforming with Pandas

Code

```
df.isnull().sum()                       # count missing values
df.fillna(0)                            # fill missing with 0
df.dropna()                             # or drop rows with NaN
df.drop_duplicates()                    # remove repeated rows
df["Age"] = df["Age"].astype(int)       # change data type
df["Date"] = pd.to_datetime(df["Date"])  # convert to datetime
df["Name"] = df["Name"].str.strip().str.title()# string cleaning
df["Grade"] = df["Marks"].apply(lambda m: "Pass" if m >= 50 else "Fail")
df.rename(columns={"Marks": "Score"}, inplace=True)
df["City"].value_counts()               # frequency of each category
df.corr(numeric_only=True)              # correlation matrix
df.to_csv("clean.csv", index=False)     # save the cleaned data
```

Method chaining and copy vs view

Pandas operations return a new DataFrame, so several steps can be **chained** into one readable expression. Here employees earning above ₹ 40,000 are selected, a 10% bonus column is created, and the top three bonuses are displayed:

Code

```
result = (emp.query("Salary > 40000")
          .assign(Bonus=lambda d: d.Salary * 0.10)
          .sort_values("Bonus", ascending=False)
          .head(3)[["Name", "Bonus"]])
print(result)
```

Output

```
   Name   Bonus
4  Divya  6200.0
2  Meena  5800.0
0   Asha  5000.0
```

Another important point is **copy versus view**. A slice such as `sub = df[df.Age > 20]` may refer to the original data, and modifying it can raise a *SettingWithCopyWarning* or change the wrong object. Use `sub = df[df.Age > 20].copy()` when an independent copy is wanted, and assign changes with `df.loc[row_condition, "col"] = value`. Also keep the original raw file unchanged and write the cleaned data to a **new file**, so the work can always be repeated.

8.5 Complete Mini-Example: End-to-End Wrangling

The following program combines the ideas of the entire unit – load, detect missing values, impute, remove an outlier by the IQR rule, encode a category and scale numeric columns.

Code

```
import numpy as np, pandas as pd
from sklearn.preprocessing import MinMaxScaler

df = pd.DataFrame({
    "Age":    [22, 25, np.nan, 30, 28, 24],
    "Salary": [30000, 35000, 32000, 400000, 38000, np.nan],   # 400000 = outlier
    "City":   ["Chennai", "Salem", "Chennai", None, "Salem", "Chennai"]})

# 1. Missing values
df["Age"]    = df["Age"].fillna(df["Age"].median())
df["Salary"] = df["Salary"].fillna(df["Salary"].median())
df["City"]   = df["City"].fillna(df["City"].mode()[0])

# 2. Outlier capping with IQR
q1, q3 = df["Salary"].quantile([0.25, 0.75])
iqr = q3 - q1
df["Salary"] = df["Salary"].clip(q1 - 1.5 * iqr, q3 + 1.5 * iqr)

# 3. Encoding
df = pd.get_dummies(df, columns=["City"], dtype=int)

# 4. Scaling
df[["Age", "Salary"]] = MinMaxScaler().fit_transform(df[["Age", "Salary"]])
```

```
print(df.round(2))
```

Output

```
   Age  Salary  City_Chennai  City_Salem
0 0.00    0.00             1           0
1 0.38    0.36             0           1
2 0.38    0.14             1           0
3 1.00    1.00             1           0
4 0.75    0.57             0           1
5 0.25    0.36             1           0
```

After the steps, the dataset has **no missing values**, the extreme salary of ₹ 4,00,000 is capped, the City column is replaced by binary columns, and Age and Salary lie between 0 and 1 – the data is **ready for a machine-learning algorithm**.

8.6 NumPy vs Pandas

Point	NumPy	Pandas
Main structure	ndarray (n-dimensional)	Series (1-D) and DataFrame (2-D)
Data type	Homogeneous – all elements same type	Heterogeneous – each column may differ
Labels	Integer positions only	Row index and column names
Missing data	Weak support (np.nan)	Strong support (isnull, fillna, dropna)
Speed and memory	Faster, uses less memory	Slightly slower, more features
Best for	Numerical computation, matrices, images	Tabular data analysis, cleaning, file handling
Reading files	Limited (loadtxt)	read_csv, read_excel, read_sql, read_json

8.7 Other Data Wrangling Tools

Tool	Type	Remark
Microsoft Excel / Google Sheets	Spreadsheet	Good for small data and quick checks
OpenRefine	Open-source GUI tool	Cleaning messy text data, clustering similar values
SQL	Query language	Filtering, joining and aggregating in the database
R (dplyr, tidyr)	Programming language	Alternative to Pandas in statistics

Tool	Type	Remark
Polars	Python library	Very fast DataFrame library written in Rust
Dask / PySpark	Distributed processing	Data larger than memory; big-data clusters
Trifacta / Alteryx / Power Query	Commercial GUI tools	Visual data-preparation workflows