Data Cleaning with Python and Pandas

1. What Is Data Cleaning?

Real-world data is rarely ready for analysis.

A dataset may contain:

- Missing values
- Duplicate rows
- Incorrect data types
- Extra spaces
- Inconsistent capitalization
- Different spellings for the same value
- Invalid dates
- Impossible numbers
- Incorrect categories
- Outliers
- Empty strings
- Incorrect column names
- Mixed formats

Data cleaning is the process of finding these problems, deciding how they should be handled, and producing data that is consistent, valid, and ready for analysis or further processing.

Pandas is commonly used for this work because its `DataFrame` structure is designed for tabular data and provides operations for inspecting, transforming, and cleaning datasets.

The important idea

Data cleaning is **not simply deleting bad rows**.

A good data-cleaning process asks:

> What does this value mean, why is it wrong or missing, and what is the safest way to handle it?

---

2. Why Data Cleaning Matters

Imagine this dataset:

```
Name      Age      City
Hafsa     25       Lahore
Asim      24       lahore
Ali       999      Islamabad
Sara      NaN      Lahore
Hafsa     25       Lahore
```

There are several problems:

- `lahore` and `Lahore` represent the same city.
- `999` is probably not a realistic age.
- Sara's age is missing.
- Hafsa appears twice.
- Data may need consistent formatting before analysis.

If you calculate statistics without cleaning the data, your results may be misleading.

For example:

`df["Age"].mean()`

A value such as `999` can dramatically distort the average.

Clean data should be:

Consistent □ Valid □ Complete enough □ Correctly typed □ Ready for use

---

3. The Data Cleaning Mental Model

A useful workflow is:

Raw Data
□
Inspect
□
Understand the problems
□
Standardize
□
Handle missing values
□
Remove or resolve duplicates
□
Fix data types
□
Validate values
□
Handle unusual values
□
Verify the cleaned dataset
□

Save cleaned data

Do not immediately start deleting rows.

Inspect first. Clean second. Validate third.

---

4. Load the Dataset

```
import pandas as pd

df = pd.read_csv("customers.csv")
```

You can also load Excel:

```
df = pd.read_excel("customers.xlsx")
```

Or JSON:

```
df = pd.read_json("customers.json")
```

---

5. Inspect Before Cleaning

The first step is understanding what you received.

View the first rows

```
df.head()
```

View the last rows

```
df.tail()
```

Check dimensions

```
df.shape
```

Example:

```
(1000, 8)
```

This means:

```
1000 rows
8 columns
```

View column names

```
df.columns
```

Check data types

```
df.dtypes
```

Get general information

```
df.info()
```

Get statistical information

```
df.describe()
```

Check unique values

```
df["City"].unique()
```

Count unique values

```
df["City"].nunique()
```

Count values

```
df["City"].value_counts()
```

`value_counts()` is particularly useful for finding inconsistent categories.

---

6. Create a Cleaning Checklist

Before modifying the dataset, identify possible problems.

For example:

```
print(df.shape)
print(df.columns)
print(df.dtypes)
print(df.isna().sum())
print(df.duplicated().sum())
```

This gives you a quick initial report.

A more detailed inspection:

```
for column in df.columns:
    print(f"\n{column}")
    print(df[column].value_counts(dropna=False).head(10))
```

This can reveal unexpected values.

---

7. Missing Values

Missing values are one of the most common data-quality problems.

Pandas can represent missing information using values such as `NaN`, `NaT`, and `pd.NA`, depending on the data type. `isna()` and `notna()` are the standard ways to detect missing values.

Example:

```
df = pd.DataFrame({
    "Name": ["Hafsa", "Asim", "Ali"],
    "Age": [25, None, 23]
})
```

---

8. Detect Missing Values

Check whether each value is missing

`df.isna()`

Count missing values per column

`df.isna().sum()`

Example:

```
Name    0
Age     1
```

Calculate the percentage of missing values

```
df.isna().mean() * 100
```

This is useful when datasets are large.

---

9. Remove Missing Rows

Use `dropna()` when removing a row is appropriate.

```
df = df.dropna()
```

This removes rows containing missing values.

Pandas' `dropna()` can operate on rows or columns.

Remove rows only when a specific column is missing

```
df = df.dropna(subset=["Name"])
```

This is often safer than removing rows because of any missing value anywhere.

---

10. Fill Missing Values

Sometimes deleting data is not appropriate.

Use `fillna()`:

```
df["Age"] = df["Age"].fillna(0)
```

But blindly using `0` can create incorrect information.

A better choice may be the median:

```
df["Age"] = df["Age"].fillna(df["Age"].median())
```

For categorical data:

```
df["City"] = df["City"].fillna("Unknown")
```

Pandas provides `fillna()` specifically for replacing missing values.

---

11. Choosing How to Handle Missing Data

There is no universal rule.

Delete the row when:

- The row is incomplete and unusable.
- Only a tiny number of rows are affected.
- The missing field is essential.

Fill with a statistic when:

- The value is numeric.
- Using mean or median makes sense.
- Removing rows would lose too much useful data.

Fill with a category when:

- A missing category has meaningful interpretation.

Example:

```
df["City"] = df["City"].fillna("Unknown")
```

Keep it missing when:

- Missingness itself carries meaning.
- You cannot safely infer the value.

---

12. Mean vs Median

Suppose:

```
Age
22
24
25
26
999
```

The mean is affected heavily by `999`.

The median is more resistant to extreme values.

```
df["Age"].fillna(df["Age"].median())
```

This does **not** mean median is always better.

The correct choice depends on the meaning and distribution of the data.

---

13. Empty Strings Are Not Always Missing Values

Consider:

```
df["Name"]
```

with values:

```
Hafsa
""
Asim
"  "
Ali
```

An empty string is a string, not necessarily a pandas missing value.

You can detect empty strings:

```
df["Name"].str.strip().eq("")
```

Convert empty strings to missing values:

```
df["Name"] = df["Name"].replace(r"^\s*$", pd.NA, regex=True)
```

Then:

```
df["Name"].isna().sum()
```

---

14. Removing Duplicate Rows

Duplicate records can cause incorrect counts and statistics.

Check duplicates:

```
df.duplicated()
```

Count them:

```
df.duplicated().sum()
```

Remove them:

```
df = df.drop_duplicates()
```

Pandas provides `duplicated()` to identify duplicates and `drop_duplicates()` to remove them.

---

15. Duplicate Detection Using Specific Columns

Suppose:

```
customer_id
name
email
```

Two rows may differ in other fields but represent the same customer.

Check using:

```
df.duplicated(subset=["email"])
```

Remove duplicates:

```
df = df.drop_duplicates(subset=["email"])
```

Keep the last record:

```
df = df.drop_duplicates(
    subset=["email"],
    keep="last"
)
```

Possible values for `keep` include:

```
"first"
"last"
False
```

---

16. Duplicate Rows vs Duplicate IDs

These are not necessarily the same problem.

Exact duplicate

```
101  Hafsa  Lahore
101  Hafsa  Lahore
```

The complete rows are identical.

Duplicate identifier

```
101  Hafsa  Lahore
101  Hafsa  Islamabad
```

The rows are different, but the ID is repeated.

Do not automatically delete the second row.

It may represent:

- Multiple transactions
- Multiple addresses
- Historical records
- A genuine data error

You need to understand the dataset first.

---

17. Clean Column Names

Raw datasets often contain:

```
Customer Name
customer_city
AGE
Email Address
```

You may want consistent names.

```
df.columns = (
    df.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", "_")
)
```

Result:

```
customer_name
customer_city
age
email_address
```

Consistent column names make code easier to maintain.

---

18. Remove Extra Spaces

Suppose:

```
"Hafsa"
" Hafsa"
"Hafsa "
"  Hafsa  "
```

These can appear to be different values.

Clean them:

```
df["Name"] = df["Name"].str.strip()
```

For every string column:

```
string_columns = df.select_dtypes(include="string").columns

for column in string_columns:
    df[column] = df[column].str.strip()
```

---

19. Standardize Capitalization

Suppose:

```
lahore
```

```
Lahore
LAHORE
```

Convert to lowercase:

```
df["City"] = df["City"].str.lower()
```

Or title case:

```
df["City"] = df["City"].str.title()
```

Result:

```
Lahore
Lahore
Lahore
```

Choose one format and apply it consistently.

---

20. Standardize Categories

Suppose a gender column contains:

```
Male
male
M
MALE
Female
female
F
```

You can map them:

```
gender_map = {
    "male": "Male",
    "m": "Male",
    "female": "Female",
    "f": "Female"
}

df["gender"] = (
    df["gender"]
    .str.strip()
    .str.lower()
    .map(gender_map)
)
```

Now the category becomes consistent.

---

21. Be Careful With `.map()`

If a value does not exist in the mapping:

```
gender_map = {
    "male": "Male",
    "female": "Female"
}
```

Then:

```
df["gender"].map(gender_map)
```

can produce missing values for unexpected categories.

That can be useful because it exposes bad data.

But always inspect the result.

---

22. Find Unexpected Categories

Use:

```
df["City"].value_counts(dropna=False)
```

Suppose you see:

```
Lahore      500
Islamabad   300
Karachi     200
Lahor        4
Lahore       2
```

The unusual spelling:

```
Lahor
```

should be investigated.

You could correct it:

```
df["City"] = df["City"].replace({
   "Lahor": "Lahore"
```

```
})
```

---

23. Numeric Data Stored as Text

Consider:

```
"100"
"250"
"300"
"unknown"
```

Pandas may treat the column as strings.

Check:

```
df["price"].dtype
```

Convert safely:

```
df["price"] = pd.to_numeric(
    df["price"],
    errors="coerce"
)
```

Invalid values become missing values.

Then inspect:

```
df["price"].isna().sum()
```

Do not silently assume that every failed conversion is safe to discard.

---

24. Remove Currency Symbols

Suppose prices look like:

```
$1,200
$850
$2,500
```

Clean them:

```
df["price"] = (
    df["price"]
    .str.replace("$", "", regex=False)
    .str.replace(",", "", regex=False)
)

df["price"] = pd.to_numeric(
    df["price"],
    errors="coerce"
)
```

Now the column can be used for calculations.

---

25. Clean Percentage Values

Suppose:

```
"20%"
"15%"
"7%"
```

You can remove `%`:

```
df["discount"] = (
    df["discount"]
    .str.replace("%", "", regex=False)
)
```

Then convert:

```
df["discount"] = pd.to_numeric(
    df["discount"],
    errors="coerce"
)
```

If you need decimal proportions:

```
df["discount"] = df["discount"] / 100
```

---

26. Clean Dates

Raw data may contain:

```
2026-01-15
15/01/2026
January 15, 2026
```

Convert to pandas datetime:

```
df["order_date"] = pd.to_datetime(
    df["order_date"],
    errors="coerce"
```

)

Invalid dates become `NaT`.

Check them:

```
df[df["order_date"].isna()]
```

`NaT` is pandas' missing-value representation for datetime-like data.

---

27. Extract Information From Dates

Once the column is datetime:

```
df["year"] = df["order_date"].dt.year
```

```
df["month"] = df["order_date"].dt.month
```

```
df["day"] = df["order_date"].dt.day
```

```
df["weekday"] = df["order_date"].dt.day_name()
```

This makes date-based analysis much easier.

---

28. Validate Numeric Ranges

Suppose age should be between 0 and 120.

Find invalid values:

```
invalid_age = df[
    (df["age"] < 0) |
    (df["age"] > 120)
]
```

Do not immediately delete them.

First inspect:

```
print(invalid_age)
```

Then decide whether they should be:

- Corrected
- Replaced with missing
- Removed
- Investigated

---

29. Validate Categories

Suppose status should only contain:

```
pending
approved
rejected
```

Check unexpected values:

```
allowed_statuses = {
    "pending",
    "approved",
```

```
    "rejected"
}

invalid_status = df[
    ~df["status"].isin(allowed_statuses)
]
```

You can inspect:

```
print(invalid_status)
```

---

30. Validate Relationships Between Columns

Data can be individually valid but logically inconsistent.

For example:

```
start_date = 2026-05-10
end_date   = 2026-05-01
```

Both are valid dates.

But the relationship is invalid.

Check:

```
invalid_dates = df[
    df["end_date"] < df["start_date"]
]
```

This is an important idea:

>

> Data validation is not only about individual values. It is also about relationships between values.

---

31. Detect Outliers

An outlier is a value that is unusually far from other observations.

Example:

```
10
12
11
13
14
5000
```

You can use summary statistics:

```
df["sales"].describe()
```

One common technique is the IQR method.

```
Q1 = df["sales"].quantile(0.25)
Q3 = df["sales"].quantile(0.75)

IQR = Q3 - Q1

lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR

outliers = df[
```

```
  (df["sales"] < lower) |
  (df["sales"] > upper)
]
```

---

32. An Outlier Is Not Automatically an Error

Suppose a store normally has orders between:

`$10 and $500`

but one customer purchases:

`$10,000`

That could be:

- A genuine large order
- A business customer
- A data-entry error
- A duplicate transaction

Do not automatically delete it.

Unusual does not mean incorrect.

---

33. Detect Impossible Values

Examples:

Age = -5
Quantity = -10
Discount = 300%
Temperature = 9000°C

These deserve investigation.

Example:

```
df.loc[df["quantity"] < 0, "quantity"]
```

For a column where negatives are impossible:

```
df = df[df["quantity"] >= 0]
```

Only do this when the business rules actually say negative values are invalid.

---

34. Cleaning Text With `str`

Pandas provides vectorized string operations through `.str`. These operations can clean and transform entire columns efficiently.

Useful operations include:

```
.str.strip()
.str.lower()
.str.upper()
.str.title()
.str.replace()
.str.contains()
.str.startswith()
.str.endswith()
```

.str.len()

Example:

```
df["email"] = (
    df["email"]
    .str.strip()
    .str.lower()
)
```

---

35. Clean Email Values

Example:

```
" Hafsa@Example.COM "
```

Clean:

```
df["email"] = (
    df["email"]
    .str.strip()
    .str.lower()
)
```

You can perform a basic validation:

```
invalid_email = ~df["email"].str.contains(
    "@",
    na=False
)
```

For production systems, email validation may require stricter rules.

---

36. Standardize Boolean Values

Raw data may contain:

```
Yes
yes
Y
TRUE
1
No
N
false
0
```

Create a standard mapping:

```
boolean_map = {
    "yes": True,
    "y": True,
    "true": True,
    "1": True,
    "no": False,
    "n": False,
    "false": False,
    "0": False
}
```

Then:

```
df["active"] = (
    df["active"]
    .astype("string")
```

```
    .str.strip()
    .str.lower()
    .map(boolean_map)
)
```

---

37. Fix Column Data Types

Inspect:

```
df.dtypes
```

Convert integer-like data:

```
df["age"] = pd.to_numeric(
    df["age"],
    errors="coerce"
)
```

Convert dates:

```
df["date"] = pd.to_datetime(
    df["date"],
    errors="coerce"
)
```

Convert strings:

```
df["name"] = df["name"].astype("string")
```

Correct data types make later operations more reliable.

---

38. Why Data Types Matter

Imagine:

```
price = "100"
quantity = "5"
```

These are strings.

This:

```
price * quantity
```

does not perform normal numeric multiplication.

After conversion:

```
price = 100
quantity = 5
```

you can calculate:

```
price * quantity
```

Correct types help prevent subtle bugs.

---

39. Cleaning a Complete Dataset

Consider:

```
df = pd.DataFrame({
    "Name": [" Hafsa ", "Asim", "Hafsa"],
    "Age": ["25", "24", "25"],
    "City": ["lahore", "Islamabad ", "Lahore"],
    "Email": [
        " HAFSA@example.com ",
        "asim@example.com",
        "HAFSA@example.com"
    ]
})
```

Step 1: Clean column names

```
df.columns = (
    df.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", "_")
)
```

Step 2: Clean text

```
for column in ["name", "city", "email"]:
    df[column] = df[column].astype("string").str.strip()
```

Step 3: Standardize case

```
df["city"] = df["city"].str.title()
df["email"] = df["email"].str.lower()
```

Step 4: Convert age

```
df["age"] = pd.to_numeric(
    df["age"],
    errors="coerce"
)
```

Step 5: Detect duplicates

```
df.duplicated(subset=["email"]).sum()
```

Step 6: Remove duplicates if appropriate

```
df = df.drop_duplicates(
    subset=["email"]
)
```

---

40. A Reusable Cleaning Pipeline

Instead of writing random transformations, organize the process.

```
import pandas as pd

df = pd.read_csv("customers.csv")

# 1. Standardize column names
df.columns = (
    df.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", "_")
)

# 2. Clean text
for column in ["name", "city", "email"]:
    if column in df.columns:
        df[column] = (
            df[column]
            .astype("string")
```

```
            .str.strip()
        )

# 3. Standardize categories
if "city" in df.columns:
    df["city"] = df["city"].str.title()

# 4. Convert numeric values
if "age" in df.columns:
    df["age"] = pd.to_numeric(
        df["age"],
        errors="coerce"
    )

# 5. Convert dates
if "signup_date" in df.columns:
    df["signup_date"] = pd.to_datetime(
        df["signup_date"],
        errors="coerce"
    )

# 6. Remove exact duplicates
df = df.drop_duplicates()

# 7. Inspect missing values
print(df.isna().sum())

# 8. Inspect final data
print(df.info())
```

This is much easier to maintain than scattered transformations throughout a notebook.

---

41. Cleaning vs Validation

These concepts are related but different.

Cleaning

Changes the data.

```
df["city"] = df["city"].str.title()
```

Validation

Checks whether the data is acceptable.

```
invalid = ~df["city"].isin(
    ["Lahore", "Islamabad", "Karachi"]
)
```

Think of it as:

```
Cleaning = Fix
Validation = Check
```

A professional data pipeline often needs both.

---

42. Keep Raw Data Separate

A very important rule:

> Never destroy your only copy of the raw dataset.

Instead:

```
raw_df = pd.read_csv("customers.csv")

df = raw_df.copy()
```

Perform cleaning on `df`.

This gives you:

```
raw_df  original data
df      cleaned data
```

You can compare them if something goes wrong.

---

43. Build Cleaning Functions

If you repeatedly clean the same kind of dataset, use functions.

```
def clean_customers(df):
    df = df.copy()

    df.columns = (
        df.columns
        .str.strip()
        .str.lower()
        .str.replace(" ", "_")
    )

    df["name"] = (
        df["name"]
```

```
        .astype("string")
        .str.strip()
    )

    df["email"] = (
        df["email"]
        .astype("string")
        .str.strip()
        .str.lower()
    )

    df["age"] = pd.to_numeric(
        df["age"],
        errors="coerce"
    )

    df = df.drop_duplicates(
        subset=["email"]
    )

    return df
```

Then:

```
clean_df = clean_customers(raw_df)
```

This makes the process reusable.

---

44. Why `.copy()` Matters

Suppose you want to create a cleaned dataset:

```
clean_df = raw_df.copy()
```

Now you can safely transform `clean_df` without intending to alter the original dataset.

It also makes your intention clear:

```
raw data → preserved
clean data → modified
```

---

45. Validate After Cleaning

Never assume the cleaning worked.

Check:

```
print(clean_df.shape)

print(clean_df.isna().sum())

print(clean_df.duplicated().sum())

print(clean_df.dtypes)

print(clean_df.describe())
```

Check categories:

```
print(clean_df["city"].value_counts())
```

---

46. Use Assertions

Assertions can turn assumptions into automated checks.

```
assert clean_df["age"].ge(0).all()
```

For a maximum age:

```
assert clean_df["age"].le(120).all()
```

Check duplicate emails:

```
assert not clean_df["email"].duplicated().any()
```

Check required columns:

```
required = {
    "name",
    "email",
    "age"
}

assert required.issubset(clean_df.columns)
```

If an assertion fails, your pipeline immediately tells you that an assumption was violated.

---

47. Data Cleaning Should Be Reproducible

Imagine manually cleaning a CSV in Excel every Monday.

You might:

1. Delete duplicates

2. Fix capitalization
3. Remove spaces
4. Convert dates
5. Fix invalid values
6. Save the file

This works once.

But repeating it manually creates opportunities for mistakes.

A Python cleaning script can perform the same process consistently.

Raw CSV

Python script

Cleaning rules

Validation

Clean CSV

---

48. Data Cleaning in a Real Project

Imagine haas.dev stores resource information:

id
title
category
difficulty
author

published_date

Raw data might contain:

```
1, " Python Basics ", python, beginner, "Hafsa ", 2026-01-10
2, "API Design", Python, Beginner, "Hafsa", 2026-01-11
3, "", python, beginner, Hafsa, invalid-date
4, "API Design", Python, Beginner, Hafsa, 2026-01-11
```

Problems include:

- Extra spaces
- Inconsistent capitalization
- Missing title
- Invalid date
- Duplicate resource

A cleaning pipeline could standardize these values before the data reaches the application.

---

49. Example haas.dev Cleaning Pipeline

```
import pandas as pd

resources = pd.read_csv("resources.csv")

resources.columns = (
    resources.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", "_")
)
```

```
resources["title"] = (
    resources["title"]
    .astype("string")
    .str.strip()
)

resources["category"] = (
    resources["category"]
    .astype("string")
    .str.strip()
    .str.lower()
)

resources["difficulty"] = (
    resources["difficulty"]
    .astype("string")
    .str.strip()
    .str.lower()
)

resources["author"] = (
    resources["author"]
    .astype("string")
    .str.strip()
)

resources["published_date"] = pd.to_datetime(
    resources["published_date"],
    errors="coerce"
)

resources = resources.replace(
    r"^\s*$",
```

```
    pd.NA,
    regex=True
)

resources = resources.dropna(
    subset=["title"]
)

resources = resources.drop_duplicates(
    subset=["title"]
)

allowed_difficulties = {
    "beginner",
    "intermediate",
    "advanced"
}

invalid = ~resources["difficulty"].isin(
    allowed_difficulties
)

print("Invalid difficulty values:")
print(resources.loc[invalid])

print("Missing values:")
print(resources.isna().sum())
```

The goal is not simply to make the table look clean.

The goal is to make the data **trustworthy enough for the next stage of the system**.

---

50. Save the Clean Dataset

Once the data has been cleaned and validated:

```
clean_df.to_csv(
    "customers_clean.csv",
    index=False
)
```

Excel:

```
clean_df.to_excel(
    "customers_clean.xlsx",
    index=False
)
```

---

51. Keep a Cleaning Log

For important datasets, record what happened.

Example:

```
cleaning_report = {
    "original_rows": len(raw_df),
    "final_rows": len(clean_df),
    "duplicates_removed": (
        len(raw_df) - len(clean_df)
    ),
    "missing_values": (
        clean_df.isna().sum().to_dict()
    )
```

```
}

print(cleaning_report)
```

This makes your pipeline easier to audit.

---

52. A Practical Cleaning Checklist

Before considering a dataset clean, ask:

Structure

- Are column names correct?
- Are required columns present?
- Are there unexpected columns?

Missing data

- Which columns contain missing values?
- How much data is missing?
- Should missing values be deleted, filled, or preserved?

Duplicates

- Are there exact duplicate rows?
- Are unique identifiers duplicated?
- Do duplicate records represent real events?

Text

- Are spaces consistent?
- Is capitalization consistent?
- Are categories standardized?

Numbers

- Are numeric columns actually numeric?
- Are negative values valid?
- Are values within expected ranges?
- Are there suspicious outliers?

Dates

- Are dates correctly parsed?
- Are invalid dates present?
- Are date relationships logical?

Validation

- Do business rules hold?
- Are required fields present?
- Are categories valid?

Output

- Did you preserve the raw data?
- Did you validate after cleaning?
- Did you save the cleaned dataset correctly?

---

53. Common Data Cleaning Mistakes

Mistake 1: Deleting every missing row

```
df = df.dropna()
```

This may remove too much useful data.

Mistake 2: Replacing every missing number with zero

`df = df.fillna(0)`

Missing does not necessarily mean zero.

Mistake 3: Removing every outlier

An outlier can be a legitimate observation.

Mistake 4: Fixing values without understanding them

Changing:

`Lahor → Lahore`

may be reasonable.

Changing an unusual customer name or transaction amount automatically may not be.

Mistake 5: Cleaning only what looks visually wrong

Some problems are invisible until you analyze:

- Data types
- Relationships
- Duplicate IDs

- Invalid ranges
- Missing values
- Unexpected categories

Mistake 6: Modifying raw data directly

Always keep the original dataset.

Mistake 7: Cleaning without validation

A script can successfully run and still produce incorrect data.

---

54. Best Practices

1. Inspect first

Understand the dataset before modifying it.

2. Preserve raw data

Keep the original source untouched.

3. Make cleaning rules explicit

Instead of mysterious transformations:

```
df["age"] = df["age"].fillna(...)
```

know why that value is being used.

4. Prefer reproducible code

A script is easier to repeat than manual editing.

5. Validate after transformations

Check that your assumptions still hold.

6. Treat domain knowledge as important

Python can detect that:

`age = 999`

is unusual.

Only domain knowledge can tell you what the correct value should be.

7. Keep cleaning separate from analysis

First prepare trustworthy data.

Then analyze it.

---

55. Cleaning Pipeline vs One-Off Cleaning

One-off analysis

A notebook may be enough:

```
df = df.drop_duplicates()
df["city"] = df["city"].str.title()
```

Repeated production workflow

Use reusable functions or modules:

```
project/
│
├── data/
│   ├── raw/
│   └── cleaned/
│
├── src/
│   └── cleaning.py
│
├── tests/
│
└── main.py
```

This turns data cleaning into a maintainable part of the application.

---

56. Mini Project: Customer Data Cleaner

Create:

```
customers.csv
```

with columns:

```
customer_id
name
```

email
age
city
signup_date

Intentionally include:

- Missing ages
- Duplicate emails
- Extra spaces
- Different capitalization
- Invalid ages
- Invalid dates
- Empty names
- Duplicate rows

Your program should:

1. Load the dataset.
2. Preserve the raw data.
3. Standardize column names.
4. Clean text.
5. Convert numeric values.
6. Convert dates.
7. Detect missing values.
8. Handle missing values appropriately.
9. Detect duplicate customers.
10. Validate age.
11. Validate dates.
12. Print a cleaning report.
13. Save the cleaned dataset.

Expected flow:

```
customers.csv
    
Load
    
Inspect
    
Clean
    
Validate
    
Report
    
customers_clean.csv
```

---

57. 5 Minute Challenge

Given:

```
df = pd.DataFrame({
    "Name": [" Hafsa", "ASIM ", "Hafsa"],
    "Age": ["25", "24", "25"],
    "City": ["lahore", "ISLAMABAD", "Lahore"],
    "Email": [
        "HAFSA@example.com",
        "asim@example.com",
        "HAFSA@example.com"
    ]
})
```

Clean it so that:

- Names have no surrounding spaces.
- Names use consistent capitalization.
- Cities use consistent capitalization.
- Emails are lowercase.
- Ages are numeric.
- Duplicate emails are removed.

Try solving it without looking at the previous examples.

---

58. Exercises

Exercise 1

Create a DataFrame containing:

```
Name
Age
City
```

Add missing values and identify them.

Exercise 2

Create a dataset containing:

```
Lahore
lahore
LAHORE
 Lahore
```

Standardize all values.

Exercise 3

Create a price column:

```
$100
$1,250
$500
unknown
```

Convert it to numeric values.

Exercise 4

Create a dataset with duplicate emails.

Find duplicate customers and keep only the first record.

Exercise 5

Create an age column containing:

```
20
25
-3
35
200
```

Identify invalid ages.

Exercise 6

Create a date column containing:

```
2026-01-01
2026-02-15
not-a-date
2026-03-20
```

Convert it using `pd.to_datetime()` and identify invalid values.

Exercise 7

Build a complete cleaning function:

```
def clean_data(df):
    ...
    return df
```

Your function should:

- Standardize column names
- Strip whitespace
- Convert numeric columns
- Convert dates
- Remove duplicates
- Validate important fields

---

59. Mini Quiz

Question 1

What is the main purpose of data cleaning?

A. Make code shorter
B. Make data trustworthy and consistent
C. Increase the number of rows
D. Delete all missing values

Answer: B

Question 2

Which method detects missing values?

A. `duplicated()`
B. `isna()`
C. `sort_values()`
D. `astype()`

Answer: B

Question 3

Which method removes duplicate rows?

A. `dropna()`
B. `fillna()`
C. `drop_duplicates()`
D. `remove()`

Answer: C

Question 4

What does this do?

```
pd.to_numeric(df["age"], errors="coerce")
```

A. Sorts ages
B. Converts values to numeric and converts invalid values to missing
C. Removes duplicate ages
D. Converts numbers to strings

Answer: B

Question 5

Should every outlier be deleted?

Answer: No.

An outlier may be a valid observation.

---

60. Interview Questions

Beginner

1. What is data cleaning?

Preparing raw data by identifying and handling problems such as missing values, duplicates, inconsistent formats, invalid values, and incorrect data types.

2. How do you find missing values in pandas?

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

3. How do you remove duplicate rows?

```
df.drop_duplicates()
```

4. How do you convert a column to numeric?

```
pd.to_numeric(df["column"], errors="coerce")
```

5. How do you convert a column to datetime?

```
pd.to_datetime(df["column"], errors="coerce")
```

Intermediate

6. Why shouldn't you automatically delete missing rows?

Because doing so can remove useful information and introduce bias.

7. Why is `errors="coerce"` useful?

It allows invalid values to become missing values so they can be detected and handled rather than causing the conversion to fail.

8. How do you identify duplicate customers using email?

```
df.duplicated(subset=["email"])
```

9. How can you standardize text values?

Using methods such as:

```
.str.strip()
.str.lower()
.str.upper()
.str.title()
```

10. What is the difference between cleaning and validation?

Cleaning modifies data to improve its quality. Validation checks whether the resulting data satisfies expected rules.

Advanced

11. Why should raw data be preserved?

So the original source remains available for auditing, debugging, comparison, and reprocessing.

12. Why can an outlier not automatically be considered an error?

Because unusual values can represent legitimate real-world events.

13. How can data cleaning be made reproducible?

By implementing transformations as reusable functions, scripts, or pipeline stages.

14. Why is domain knowledge important in data cleaning?

Because software can detect unusual patterns, but the meaning of a value often depends on the business or application domain.

---

61. Data Cleaning Cheat Sheet

Inspect

```
df.head()
df.shape
df.info()
df.dtypes
df.describe()
df.columns
df["column"].unique()
df["column"].value_counts(dropna=False)
```

Missing values

```
df.isna()
df.isna().sum()
df.dropna()
df.fillna(value)
```

Duplicates

```
df.duplicated()
df.duplicated().sum()
```

```
df.drop_duplicates()
df.drop_duplicates(subset=["email"])
```

Strings

```
df["name"].str.strip()
df["name"].str.lower()
df["name"].str.upper()
df["name"].str.title()
df["name"].str.replace(...)
```

Numeric conversion

```
pd.to_numeric(
    df["price"],
    errors="coerce"
)
```

Dates

```
pd.to_datetime(
    df["date"],
    errors="coerce"
)
```

Validation

```
df["age"].between(0, 120)
df["status"].isin(allowed_values)
```

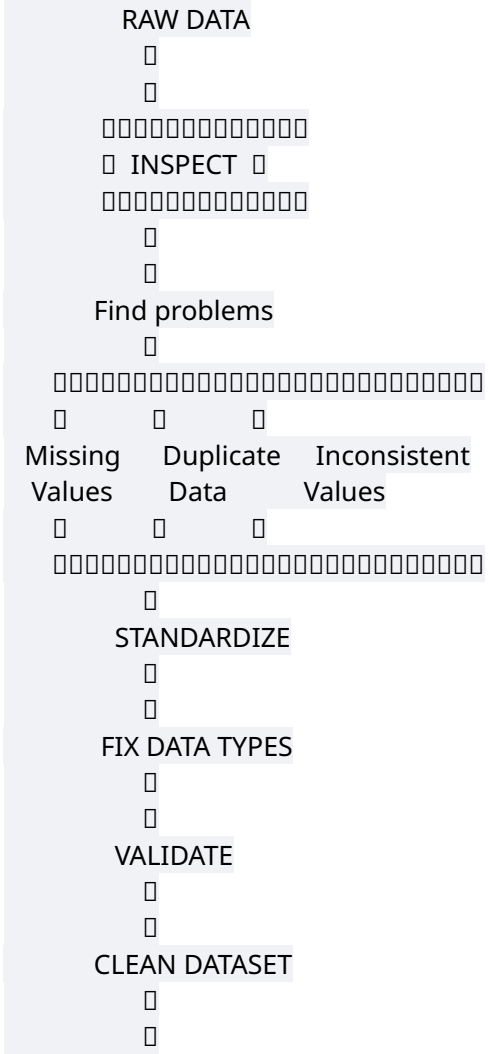
Save

```
df.to_csv(
    "cleaned.csv",
    index=False
)
```

---

62. The Professional Mental Model

Think of data cleaning as a pipeline:



```
ANALYZE
```

The important shift is:

> **Do not clean data because it looks messy. Clean it according to explicit rules about what valid data means.**

---

63. Key Takeaways

1. Real-world datasets are usually messy.
2. Data cleaning makes datasets consistent and usable.
3. Always inspect data before modifying it.
4. Missing values require decisions, not automatic deletion.
5. Duplicate rows and duplicate identifiers are different problems.
6. Text should often be standardized.
7. Numeric and date columns should have appropriate data types.
8. Invalid values should be detected through validation rules.
9. Outliers are not automatically errors.
10. Preserve the raw dataset.
11. Make cleaning reproducible.
12. Validate after cleaning.
13. Domain knowledge is essential.
14. A clean dataset is not necessarily a correct dataset until it has been validated.
15. Good data cleaning prepares the data for reliable analysis, visualization, machine learning, or application logic.

Official References

Pandas documentation:

https://pandas.pydata.org/docs/

Pandas User Guide:
https://pandas.pydata.org/docs/user_guide/

Pandas Missing Data Guide:
https://pandas.pydata.org/docs/user_guide/missing_data.html

Pandas Indexing and Duplicate Data Guide:
https://pandas.pydata.org/docs/user_guide/indexing.html

Pandas Getting Started Tutorials:
https://pandas.pydata.org/docs/getting_started/intro_tutorials/

These references correspond to the current pandas 3.0.6 documentation.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

Resources:

https://dev-roast-app.vercel.app/resources