Essential Guide to Clean Data

Clean your data, clarify your insights: practical techniques for reliable analysis.

A hands-on guide to cleaning and validating data with Python for reliable analysis.

rubió
Metabolomics

Ibon Martínez-Arranz | imartinez@labrubiocom
Data Science Manager at Rubió Metabolomics
www.rubiometabolomics.com

Ibon Martínez-Arranz, PhD in Mathematics and Statistics, holds MSc degrees in Applied Statistical Techniques and Mathematical Modeling. He has extensive expertise in statistical modeling and advanced data analysis techniques. Since 2017, he has led the Data Science area at Rubió Metabolomics, driving predictive model development and statistical computing management for metabolomics, data handling, and R&D projects. His doctoral research in Mathematics investigated and adapted genetic algorithms to improve the classification of NAFLD subtypes.

Itziar Mincholé Canals | iminchole@labrubio.com
Data Specialist at Rubió Metabolomics
www.rubiometabolomics.com

Itziar Mincholé Canals has a degree in physical sciences (1999) from the University of Zaragoza (Spain). In 1999 she began developing her professional career as a software analyst and developer, mainly working on web development projects and database programming in different sectors. In 2009 she joined OWL Metabolomics as a bioinformatician, where she has participated in several R&D projects and has developed several software tools for metabolomics. In 2016 she completed the master's degree in applied statistics with R software. After joining the Data Science department, she also supports data analysis.

Essential Guide to Clean Data

Ibon Martínez-Arranz

Life Feels Good

rubió Metabolomics

Contents

Essential Guide to Clean Data

Introduction — Why Clean Data Matters

Before we even think about training a model, generating a plot, or calculating a correlation, there's one thing that every data scientist, analyst, or researcher needs to do: clean the data.

Cleaning data might not be the most glamorous part of data science, but it is undoubtedly one of the most important. It's the foundation on which everything else is built. No matter how advanced your algorithms are or how elegant your visualizations look, if your data is full of errors, inconsistencies, and missing values, your results will be misleading at best—and completely wrong at worst.

What Do We Mean by "Clean Data"?

"Clean data" is a bit of a vague term, but broadly speaking, it refers to data that:

- Has consistent formatting and structure
- Uses the correct data types (e.g., dates are stored as dates, numbers as numbers)
- Has no missing or malformed values—or at least, missing data has been handled appropriately
- Doesn't contain obvious errors, duplicates, or outliers (unless justified)
- Matches across sources if needed (e.g., foreign keys between tables)

- Is well-documented and reproducible

Clean data isn't perfect—it's realistic, usable, and trustworthy. In short, clean data is **fit for purpose**.

Dirty Data in the Real World

Unfortunately, most real-world datasets are messy. Very messy. Columns might be misnamed, data might be missing without explanation, units might be inconsistent, and errors can sneak in during data entry, export, or transformation. In clinical datasets, it's common to see blood pressure measured in mmHg mixed with values that look suspiciously like they came from a different scale—or simply typos.

This messiness isn't just annoying—it's dangerous. As noted by Karr et al. (2006), poor data quality can invalidate scientific results, introduce bias, and waste enormous amounts of time and resources. Especially in healthcare, misleading conclusions from dirty data can have real-world consequences.

The Hidden Cost of Not Cleaning

Skipping the cleaning process—or doing it half-heartedly—has cascading effects:

- **Garbage in, garbage out**: Bad input leads to bad output. Your model may appear to perform well, but it's learning from flawed assumptions.
- **Reproducibility issues**: Without clear cleaning steps, it becomes hard to reproduce results.
- **Loss of trust**: Colleagues and stakeholders lose confidence in your analysis if inconsistencies pop up.
- **Compliance risks**: In regulated environments like clinical trials, dirty data violates standards such as GCP (Good Clinical Practice) and ALCOA+ principles.

Data Cleaning and ALCOA+

ALCOA+ stands for:

- **Attributable**: It should be clear who recorded the data.
- **Legible**: The data should be readable and permanent.
- **Contemporaneous**: Recorded at the time of the event.
- **Original**: The data must be the source or a certified copy.
- **Accurate**: Free from error.

The "+" includes: Complete, Consistent, Enduring, and Available. Data cleaning isn't just about technical correctness—it's about meeting these expectations too.

This Book's Approach

This book is hands-on. We're going to walk through real problems and clean them using Python—mostly with pandas, but also with other useful tools. Each chapter tackles a typical problem:

- What to do with missing data?
- How do I handle weird date formats?
- Are outliers always wrong?
- How do I track what changes I made to the data?

We'll keep things practical, reproducible, and honest. You'll get code examples, visualizations, and plenty of tips based on real-world experience.

References

- Karr, A. F., Sanil, A. P., & Banks, D. L. (2006). Data quality: A statistical perspective. *Statistical Methodology*, 3(2), 137–173. https://doi.org/10.1016/j.stamet

.2005.08.005

- EMA (2010). Reflection paper on expectations for electronic source data and data transcribed to electronic data collection tools in clinical trials. *European Medicines Agency*. https://www.ema.europa.eu
- FDA (2018). *Data Integrity and Compliance With Drug CGMP: Questions and Answers*. https://www.fda.gov/media/119267/download

Loading and Inspecting Raw Data

Once you get your hands on a dataset, the first instinct is often to run a model or generate a plot. But before doing anything fancy, it's crucial to slow down and look at the data you've got. Not the idea of the data. The actual contents.

Load it right

In Python, we commonly use `pandas.read_csv()` to load tabular data. But things can get tricky fast. Maybe the file uses semicolons instead of commas. Maybe there's an encoding problem. Maybe the headers are actually in row 3. Loading data is often the first sign that something isn't quite right.

```
import pandas as pd

# Default load
df = pd.read_csv("data.csv")

# Custom separator and encoding
df = pd.read_csv("data.csv", sep=';', encoding='latin1')
```

Some useful arguments in `read_csv()`:

- `sep`, `delimiter`: For files using `;` or `\t`
- `encoding`: Try `'utf-8'`, `'latin1'`, or `'ISO-8859-1'` if you see encoding errors
- `skiprows`: If the header isn't in the first line
- `na_values`: To specify what strings should be treated as missing

First visual inspection

Once loaded, start simple:

```
df.head()
df.tail()
df.sample(5)
```

Then go deeper:

```
df.info()
df.describe()
df.columns
df.dtypes
```

These commands tell you:

- How many rows and columns there are
- If any columns are missing values
- What data types are being used
- If numerical columns have suspicious max/min values

Common surprises

At this stage, you often spot things like:

- IDs read as float instead of string (e.g., `00123` becomes `123.0`)
- Columns with mixed types (e.g., numbers and text)
- Dates stored as object instead of datetime
- Missing values coded as “n/a”, “NA”, or “-”

These are all signals that the dataset needs attention before any analysis can begin.

Save a copy

Always save the raw version and keep your cleaning steps separate. This is good scientific practice, but also aligns with GCP and ALCOA+ principles: traceability, reproducibility, and transparency.

```
df.to_csv("cleaned/step0_raw_copy.csv", index=False)
```

This way, you can always go back to the original if something goes wrong later.

Summary

- Always inspect the data before doing anything else.
- Be careful with loading parameters—files can be messy.
- Print and scan the first few rows, check data types and summaries.
- Save a copy of the raw data before you begin cleaning.

We haven't modified anything yet—but now we know what we're dealing with. Let's get our hands dirty in the next chapter.

Handling Missing Values

Missing values are everywhere. In clinical data, you might have patients who skipped visits. In survey data, some respondents might refuse to answer certain questions. In IoT devices, a sensor might temporarily go offline. Whatever the source, missing data is a reality—and handling it well is a critical skill.

Why missingness matters

The way we handle missing data can have a big impact on the conclusions we draw. Drop too much, and you may lose signal or bias your sample. Impute the wrong way, and you risk creating false confidence. There's no universal solution, but there are strategies that work well in different situations.

Let's say you're building a model to predict blood glucose levels. If half your rows are missing BMI values, dropping those rows could eliminate important patterns. On the other hand, filling them all with the average might mask real variability. Every decision here influences your final model.

Types of missing data

Statisticians typically distinguish three types:

- **MCAR (Missing Completely At Random)**: There's no pattern in the missingness. This is ideal (and rare).

- **MAR (Missing At Random)**: The missingness depends on observed variables.
- **MNAR (Missing Not At Random)**: The missingness depends on unobserved variables.

For example:

- MCAR: A lab machine randomly failed to record values.
- MAR: Older patients are less likely to answer a digital survey.
- MNAR: Patients with very high blood pressure skipped follow-up visits.

Understanding this classification helps you decide what methods are safe to use.

Detecting missing values in pandas

```
df.isna().sum()
df[df['column_name'].isna()]
```

To get a quick overview:

```
missing_percent = df.isna().mean().round(2) * 100
print(missing_percent.sort_values(ascending=False))
```

You can also visualize missingness with the `missingno` library:

```
import missingno as msno
msno.matrix(df)
msno.heatmap(df)
```

This shows which rows and columns have gaps and whether patterns exist. Heatmaps are great for spotting if two variables tend to be missing together.

Strategies to handle missing values

Drop them (but be careful)

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

This works if only a few rows are affected and the loss of data won't bias the outcome. But if 30% of your dataset disappears, maybe it's time to pause.

You can also drop only rows missing specific columns:

```
df = df.dropna(subset=['age', 'sex'])
```

Or drop columns with too many missing values:

```
df = df.dropna(axis=1, thresh=len(df) * 0.5)  # keep
    columns with >50% non-null
```

Impute them (simple methods)

```
df['age'].fillna(df['age'].median(), inplace=True)
df['sex'].fillna('Unknown', inplace=True)
```

Be cautious with mean imputation—it can distort distributions, especially with skewed data. Median is often a safer bet.

Impute them (fancy methods)

You can use KNN imputation, regression models, or multiple imputation packages. One example with `sklearn`:

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=5)
```

```
df[['age', 'bmi']] = imputer.fit_transform(df[['age', '
    bmi']])
```

Or with `IterativeImputer` (multivariate regression):

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

imp = IterativeImputer(random_state=42)
df_imputed = imp.fit_transform(df.select_dtypes(include='
    number'))
```

These techniques try to "guess" missing values using other variables, which is powerful—but make sure to cross-validate your models to avoid overfitting to imputed data.

Flag them

Another trick is to add an indicator column:

```
df['age_missing'] = df['age'].isna()
```

This way, your model can learn from the fact that data was missing, which might itself be informative. For example, patients who skipped lab tests might differ systematically from those who didn't.

Real-world example: BMI in a clinical dataset

Suppose you're analyzing data from a metabolic clinic and notice 15% of the patients are missing BMI. You:

1. Create a histogram of BMI to check its distribution.
2. Decide to fill missing values with the median.
3. Add a column `bmi_missing = True/False`.

4. Log this step and save an intermediate file.

```
bmi_median = df['bmi'].median()
df['bmi_missing'] = df['bmi'].isna()
df['bmi'].fillna(bmi_median, inplace=True)
df.to_csv("cleaned/step1_bmi_imputed.csv", index=False)
```

Now you've preserved the information, filled the blanks, and left a trail others can follow.

Documentation is key

Don't just fill in blanks—write down what you did and why. This step matters for ALCOA+ and for reproducibility. Also, store your imputations in separate scripts or notebook cells so they can be reviewed. If someone else picks up your project, they should be able to trace every change.

Summary

- Missing data is normal, but must be handled thoughtfully.
- Understand the type of missingness before choosing a method.
- Use pandas or missingno to detect and explore missing values.
- Decide whether to drop, impute, or flag—and document your choices.
- Be cautious with automated imputations: always think before you fill.

Handling missing values well is a small step that can make a huge difference later. Your future self—and anyone reviewing your analysis—will thank you for being careful here.

Fixing Data Types and Formats

Getting data into the right format isn't glamorous, but it's essential. If your dates are stored as strings, your categories are treated as free text, and your numbers come in as objects, you're going to hit a wall. Models won't train. Visualizations will fail. Merges will break.

Correcting data types is one of those silent but powerful steps that makes everything else work.

Why it matters

Every column in a dataset has a type: integer, float, string, datetime, boolean, categorical… but when data is loaded from CSVs, Excel files, or databases, pandas has to guess what type each column is. And it often guesses wrong.

For example:

- `object` type might actually be a date, a number, or a categorical variable.
- IDs like `00123` might be turned into `123.0`, losing leading zeros.
- `True`/`False` values might show up as strings: "Yes", "no", "TRUE", "0".

The earlier you detect and fix these issues, the smoother everything else becomes.

Checking types in pandas

```
df.dtypes
```

To get more detail:

```
print(df.info())
```

You'll see which columns are `object`, `float64`, `int64`, `bool`, `datetime64`, and so on.

Fixing numeric columns

Sometimes numbers are stored as strings:

```
df['weight'] = pd.to_numeric(df['weight'], errors='coerce')
```

The `errors='coerce'` option turns non-numeric strings (like "n/a") into `NaN`, which you can deal with later.

If you're mixing commas and dots:

```
df['height'] = df['height'].str.replace(',', '.').astype(float)
```

Converting to datetime

Dates are one of the most common troublemakers. They often come in weird formats.

```
df['visit_date'] = pd.to_datetime(df['visit_date'], errors='coerce')
```

You can specify the format if needed:

```
df['visit_date'] = pd.to_datetime(df['visit_date'],
    format='%d/%m/%Y')
```

Check for `NaT` (missing datetimes) and make sure all values were converted correctly.

To extract parts of a date:

```
df['year'] = df['visit_date'].dt.year
df['month'] = df['visit_date'].dt.month
df['weekday'] = df['visit_date'].dt.day_name()
```

Handling categorical data

Free-text categories are a pain. They lead to duplicates and inconsistencies:

```
print(df['sex'].unique())
```

You might see: `['Male', 'male', 'M', 'F', 'Female', 'f', '']`

First, clean and unify:

```
df['sex'] = df['sex'].str.strip().str.lower()
df['sex'] = df['sex'].replace({'male': 'M', 'female': 'F'
    , 'm': 'M', 'f': 'F'})
```

Then convert to category:

```
df['sex'] = df['sex'].astype('category')
```

This saves memory and tells pandas that it's not just a string—it's a label with a finite set of values.

Boolean conversions

```
df['is_smoker'] = df['is_smoker'].map({'yes': True, 'no': False})
```

Or more safely:

```
df['is_smoker'] = df['is_smoker'].str.lower().map(lambda x: x in ['yes', 'true', '1'])
```

Dealing with unit conversions

Another hidden source of trouble is mismatched units. Imagine weight in kilograms and pounds mixed in the same column.

```
# Assuming weights over 250 are probably in pounds
mask = df['weight'] > 250
df.loc[mask, 'weight'] = df.loc[mask, 'weight'] * 0.453592
```

Add a flag to track which rows were changed:

```
df['weight_converted'] = mask
```

Always document when and why you did this. Unit mismatches can break even well-trained models.

Case study: fixing a lab dataset

You receive a dataset with the following issues:

- Dates stored as text in European format
- Blood pressure as strings: “120/80”

- BMI as a mix of floats and “n/a”

Step-by-step:

```
# Convert dates
df['visit_date'] = pd.to_datetime(df['visit_date'],
    dayfirst=True, errors='coerce')

# Split blood pressure
df[['sbp', 'dbp']] = df['blood_pressure'].str.split('/',
    expand=True)
df['sbp'] = pd.to_numeric(df['sbp'], errors='coerce')
df['dbp'] = pd.to_numeric(df['dbp'], errors='coerce')

# Handle BMI
df['bmi'] = pd.to_numeric(df['bmi'], errors='coerce')
```

In less than 10 lines, you’ve fixed three major format problems and prepared the dataset for modeling or exploration.

Summary

- Always check your data types as soon as you load the data.
- Convert text to numeric, datetime, category, or boolean as needed.
- Watch for leading zeros, inconsistent formats, and strange values.
- Use `.astype()`, `pd.to_numeric()`, and `pd.to_datetime()` to take control.
- Don’t forget unit mismatches—they can be subtle but serious.

When your data types are correct, everything else becomes easier. Clean types = clean analysis.

Standardizing and Cleaning Text Data

Messy text is one of the most underestimated data problems. Unlike numbers, text is inherently flexible and ambiguous. A single column might contain values like “Male”, “male”, “MALE”, “M”, or even just “m”—and they all mean the same thing. Unless we clean and standardize that text, our models and summaries won’t understand it.

Text data cleaning is about consistency. We’re not doing NLP here—we’re just getting labels and free-form strings into shape so we can group, filter, or model them effectively.

Why text cleaning matters

Let’s say you’re analyzing clinical trial data and want to group patients by diagnosis. If the column contains:

```
Diabetes
Type 2 diabetes
T2DM
type ii diabetes
```

…you’ll get four different groups unless you normalize them.

Common problems in text data

- Inconsistent capitalization
- Trailing or leading whitespace
- Typos and abbreviations
- Accent marks and special characters
- Mixing formats (e.g., “kg” vs “kilograms”)
- Empty strings or placeholder text (“n/a”, “–”)

Basic string cleaning in pandas

```
df['diagnosis'] = df['diagnosis'].str.strip()
df['diagnosis'] = df['diagnosis'].str.lower()
df['diagnosis'] = df['diagnosis'].str.replace('-', ' ')
```

Use `.str` methods to handle most common issues:

- `str.strip()` — remove whitespace
- `str.lower()` — standardize case
- `str.replace()` — fix formatting and symbols
- `str.title()` — useful for names
- `str.contains()` — find patterns

Replacing variants and typos

If you know the most common variants:

```
mapping = {
    't2dm': 'type 2 diabetes',
    'type ii diabetes': 'type 2 diabetes',
    'diabetes type 2': 'type 2 diabetes'
```

```
}
df['diagnosis'] = df['diagnosis'].replace(mapping)
```

You can also use regex for pattern-based replacements:

```
df['code'] = df['code'].str.replace(r'[^A-Z0-9]', '',
    regex=True)
```

This removes non-alphanumeric characters, often useful in cleaning IDs.

Dealing with missing or meaningless text

Sometimes cells contain strings like “unknown”, “n/a”, or just “-”. You can treat them as missing:

```
df['notes'] = df['notes'].replace(['n/a', 'na', '-', '
    unknown'], pd.NA)
```

Once standardized, you can handle them as proper missing values.

Categorical cleanup example

You receive a column of patient-reported smoking status:

```
Yes
yes
No
no
unknown
1
0
Y
N
```

Let’s clean and standardize it:

```
def clean_smoking(value):
    val = str(value).strip().lower()
    if val in ['yes', 'y', '1', 'true']: return 'Yes'
    elif val in ['no', 'n', '0', 'false']: return 'No'
    else: return 'Unknown'

df['smoker'] = df['smoker'].apply(clean_smoking)
df['smoker'] = df['smoker'].astype('category')
```

Now you can group or analyze smoking status confidently.

Unicode and accent normalization

Text may include invisible characters or different encodings (especially in multilingual data). Normalize accents using `unicodedata`:

```
import unicodedata

def remove_accents(text):
    if isinstance(text, str):
        return ''.join(
            c for c in unicodedata.normalize('NFKD', text
                )
            if not unicodedata.combining(c)
        )
    return text

df['city'] = df['city'].apply(remove_accents)
```

Now “Málaga” and “Malaga” will match.

Summary

- Text fields are full of inconsistencies—don’t trust them blindly.

- Use `.str` methods and mappings to clean and unify values.
- Replace placeholder text like “n/a” with proper nulls.
- Normalize accents and symbols if needed.
- Always convert cleaned strings to categorical when possible.

This kind of cleaning isn’t flashy—but it’s the difference between chaos and clarity.

Detecting and Handling Duplicates

Duplicate records are like cockroaches—if you see one, there's probably more. They sneak in through system exports, user errors, or merging datasets. Sometimes duplicates are exact copies. Other times they differ by a timestamp, a typo, or an extra space.

Not all duplicates are bad. You might expect repeated lab results, or multiple visits per patient. But when duplicates are unintended, they can skew statistics, inflate sample sizes, or confuse downstream processing.

What is a duplicate?

In pandas, a duplicate row is one where all column values match another row. But you can define duplicates based on a subset of columns if needed.

Example:

```
# Exact duplicates
df.duplicated().sum()

# Drop them
df = df.drop_duplicates()
```

To check by specific columns:

```
df.duplicated(subset=['patient_id', 'visit_date']).sum()
```

Visual inspection

Start by sorting:

```
df.sort_values(by=['patient_id', 'visit_date']).head(10)
```

Sometimes duplicates are not identical but *nearly* so:

- Same patient ID and date, but different lab values
- Same name and birthdate, but different phone number

This is where your domain knowledge comes in.

Handling duplicates safely

1. Drop exact duplicates

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

You can also keep the *last* occurrence instead of the first:

```
df = df.drop_duplicates(keep='last')
```

2. Drop by key columns

```
df = df.drop_duplicates(subset=['patient_id', 'visit_date
    '])
```

Be careful—this may drop rows that are valid but repeated.

3. Aggregate near-duplicates

Sometimes it's better to collapse rows:

```
df = df.groupby(['patient_id', 'visit_date']).agg({
    'glucose': 'mean',
    'bmi': 'first'
}).reset_index()
```

You choose which fields to average, count, or take the first value from.

4. Use fuzzy matching (advanced)

When names or text fields are close but not identical, you can use `fuzzywuzzy` or `recordlinkage`. Example:

```
from fuzzywuzzy import fuzz
fuzz.ratio("John Smith", "Jon Smith")  # 96
```

This helps detect records that look like duplicates even if they're not exact matches.

Logging duplicate removal

Always log how many rows were removed and why:

```
before = len(df)
df = df.drop_duplicates()
after = len(df)
print(f"Removed {before - after} duplicate rows")
```

If working under ALCOA+, document criteria used and keep an original backup.

Case study: clinical visit log

You receive an export of visits from a hospital system. Some patients have 2 or 3 identical entries for the same date. After talking to the team, you learn it's a system glitch that repeated the same record.

```
duplicates = df[df.duplicated(subset=['patient_id', '
    visit_date'])]
print("Duplicates found:", len(duplicates))

# Drop safely
df = df.drop_duplicates(subset=['patient_id', 'visit_date
    '])
```

You document the issue, clean the dataset, and save a clean copy:

```
df.to_csv("cleaned/step2_duplicates_removed.csv", index=
    False)
```

Summary

- Duplicates can distort analysis if left unchecked.
- Use `duplicated()` and `drop_duplicates()` to detect and clean.
- Decide if duplicates are true errors or expected repetitions.
- Consider grouping and aggregating near-duplicates.
- Always log what you removed and why.

Handling duplicates is about being precise, careful, and curious. They're not just technical artifacts—they're clues to how your data was created.

Outlier Detection and Correction

Outliers are values that look suspiciously far from the rest. Maybe they're errors. Maybe they're just rare events. Either way, they can have a big impact—especially on statistics like mean and standard deviation, or models sensitive to scale.

But here's the tricky part: not all outliers are wrong. In medicine, a BMI of 60 is unusual, but it could be correct. In finance, a sudden spike in sales might reflect a real event. So before deleting or correcting anything, we need to **understand** why an outlier exists.

What is an outlier?

There's no single definition, but here are a few signs:

- A value far outside the normal range
- A point that falls outside the interquartile range (IQR)
- A value more than 3 standard deviations from the mean
- A point that breaks a business or clinical rule (e.g., heart rate > 300)

Visualizing outliers

Boxplots

```
import seaborn as sns
sns.boxplot(x=df['bmi'])
```

Boxplots are great for seeing the spread of the data and identifying points outside the whiskers.

Histograms

```
df['glucose'].hist(bins=50)
```

Helpful for spotting spikes, long tails, or gaps.

Scatter plots

Use when outliers depend on context:

```
sns.scatterplot(x='age', y='cholesterol', data=df)
```

Z-scores

For numerical columns:

```
from scipy.stats import zscore
z = zscore(df['bmi'].dropna())
outliers = df.loc[(z < -3) | (z > 3)]
```

IQR method

```
Q1 = df['bmi'].quantile(0.25)
Q3 = df['bmi'].quantile(0.75)
IQR = Q3 - Q1
```

```
4
5  outliers = df[(df['bmi'] < Q1 - 1.5 * IQR) | (df['bmi'] >
       Q3 + 1.5 * IQR)]
```

What to do with outliers

1. Investigate

- Is it a data entry error?
- Does it break known constraints?
- Could it be a valid extreme case?

2. Correct if clearly wrong

```
1  # Example: age of 250 is likely a typo
2  df.loc[df['age'] > 120, 'age'] = pd.NA
```

Or set to null to review later:

```
1  mask = df['glucose'] > 1000
2  df.loc[mask, 'glucose'] = pd.NA
```

3. Cap or clip

Limit values to a reasonable range:

```
1  df['bmi'] = df['bmi'].clip(lower=10, upper=60)
```

This preserves the row but limits the influence of the extreme value.

4. Flag them

Mark outliers for later review:

```
Q1 = df['bmi'].quantile(0.25)
Q3 = df['bmi'].quantile(0.75)
IQR = Q3 - Q1

df['bmi_outlier'] = (df['bmi'] < Q1 - 1.5 * IQR) | (df['
    bmi'] > Q3 + 1.5 * IQR)
```

5. Remove them (last resort)

Only when justified:

```
df = df[~df['bmi_outlier']]
```

Don't delete rows just because they make your plot prettier.

Real-world example: metabolic data

You're analyzing fasting glucose levels. Most values are between 70 and 120 mg/dL. But a few values are over 1000—more than ten times the typical range.

Step-by-step:

1. Plot a histogram.
2. Use IQR to define outliers.
3. Set outliers to `NaN`.
4. Add an outlier flag.
5. Document the change.

```
Q1 = df['glucose'].quantile(0.25)
Q3 = df['glucose'].quantile(0.75)
```

```
IQR = Q3 - Q1

mask = (df['glucose'] < Q1 - 1.5 * IQR) | (df['glucose']
    > Q3 + 1.5 * IQR)
df['glucose_outlier'] = mask
df.loc[mask, 'glucose'] = pd.NA
```

Summary

- Outliers can be real or errors—investigate before acting.
- Use boxplots, histograms, z-scores, or IQR to detect them.
- Consider capping, flagging, or removing—but document all steps.
- Never assume that a strange value is wrong without context.

Outliers are warning lights. They tell you where to look more closely—and sometimes, where the real story is hiding.

Resolving Inconsistencies Across Columns

Some problems aren't obvious by looking at individual columns—they live in the relationships between columns. Maybe a patient has a discharge date that comes before their admission. Or a BMI that doesn't match their height and weight. Or someone marked as deceased but with follow-up lab results.

These kinds of cross-field inconsistencies are common in real datasets. Fixing them is a bit like detective work—you need to compare, calculate, and ask "does this make sense?"

Common types of column inconsistencies

- **Date logic errors**: end date before start date, birth date in the future
- **Redundant fields mismatch**: BMI doesn't match height and weight
- **Flag contradictions**: `pregnant = True` and `sex = male`
- **Linked variables drift**: weight increased by 20 kg but height stayed the same

Logical checks with dates

Start with simple comparisons:

```
df['admit_before_discharge'] = df['admit_date'] <= df['
    discharge_date']
invalid_dates = df[~df['admit_before_discharge']]
```

Flag invalid rows and review:

```
print(invalid_dates[['patient_id', 'admit_date', '
    discharge_date']])
```

Set them to `NaT` if needed:

```
df.loc[~df['admit_before_discharge'], 'discharge_date'] =
     pd.NaT
```

Recalculating from raw data

If you have height and weight, you can recompute BMI:

```
df['height_m'] = df['height_cm'] / 100
df['bmi_calc'] = df['weight_kg'] / df['height_m']**2
```

Compare with recorded BMI:

```
bmi_diff = (df['bmi'] - df['bmi_calc']).abs()
df['bmi_mismatch'] = bmi_diff > 1.5
```

Now you can decide whether to trust the calculated or recorded value—or keep both.

Contradictory flags

Example: someone marked `pregnant = True` and `sex = male`.

```
mask = (df['pregnant'] == True) & (df['sex'] == 'M')
df.loc[mask, ['pregnant', 'sex']]
```

If it's a data entry mistake, you may want to:

- Set `pregnant = False`
- Set it to `NaN`
- Flag the row for review

Always log how many contradictions were found:

```
print("Contradictions found:", mask.sum())
```

Category mismatch

In some cases, values don't match expected category pairs:

- A "follow-up" record without an initial visit
- A "discharged" flag without an admission
- A "completed" task with a null timestamp

You can detect these with conditional checks:

```
mask = (df['visit_type'] == 'follow-up') & df['
    initial_visit_id'].isna()
df['visit_issue'] = mask
```

Value correlation

If two variables should be tightly correlated, check them:

```
sns.scatterplot(x='total_cholesterol', y='ldl')
```

If LDL is higher than total cholesterol for multiple cases, something's wrong.

You can also calculate correlation:

```
corr = df[['total_cholesterol', 'ldl']].corr()
print(corr)
```

Case study: verifying timestamps

You're working with ICU data. Each row has `start_time`, `end_time`, and `duration_minutes`.

```
# Recalculate duration
df['duration_calc'] = (df['end_time'] - df['start_time'])
    .dt.total_seconds() / 60

# Compare with recorded
df['duration_diff'] = (df['duration_calc'] - df['
    duration_minutes']).abs()
df['duration_mismatch'] = df['duration_diff'] > 5
```

Now you've spotted where duration was miscalculated, possibly from rounding errors or manual edits.

Summary

- Some errors only appear when comparing multiple columns.
- Use logic and simple math to detect contradictions.
- Recalculate values from raw inputs when possible.
- Flag inconsistencies, correct when obvious, and always document.

These aren't just bugs—they're clues. Data that doesn't make sense across columns tells you a story. Sometimes that story is about biology. Sometimes it's about a typo.

Validating and Enforcing Constraints

Data validation is about making sure the values in your dataset make sense—not just in isolation, but according to defined rules. These rules might come from your domain knowledge, regulatory standards, or logical expectations. Think of it as teaching your dataset some basic manners.

If your analysis is a house, validation is the plumbing check before you move in.

Why validation matters

Anyone can make a mistake. A user might enter a temperature in Fahrenheit instead of Celsius. A timestamp might get truncated. A value might be typed as "999" instead of "99". These small errors can cascade into bigger problems if left unchecked.

By enforcing constraints early, you catch issues before they cause confusion downstream.

Types of validations

Range checks

Ensure values fall within acceptable limits:

```
mask = (df['age'] >= 0) & (df['age'] <= 120)
df['age_invalid'] = ~mask
```

Allowed values (categorical checks)

```
valid_values = ['M', 'F']
df['sex_valid'] = df['sex'].isin(valid_values)
```

Null constraints

Make sure required fields aren't missing:

```
df['admit_date_valid'] = df['admit_date'].notna()
```

Cross-field dependencies

```
# Pregnancy flag only allowed if sex is female
mask = (df['sex'] == 'F') | (df['pregnant'] != True)
df['pregnancy_constraint_ok'] = mask
```

Pattern checks with regex

```
# Email format
pattern = r'^[\w\.-]+@[\w\.-]+\.\w{2,4}$'
df['email_valid'] = df['email'].str.match(pattern)
```

Validating with `pandera`

`pandera` is a Python library to define schemas and validate pandas DataFrames.

Example:

```
import pandera as pa
from pandera import Column, DataFrameSchema

schema = DataFrameSchema({
    "age": Column(pa.Int, checks=pa.Check.in_range(0,
        120)),
    "sex": Column(pa.String, checks=pa.Check.isin(["M", "
        F"])),
    "bmi": Column(pa.Float, nullable=True),
})

validated_df = schema.validate(df)
```

If something's wrong, it raises a clear error.

Using `pydantic` for row-level validation

If your data comes row by row (e.g., from an API), `pydantic` is a great option.

```
from pydantic import BaseModel, Field, validator

class Patient(BaseModel):
    age: int = Field(..., ge=0, le=120)
    sex: str

    @validator("sex")
    def check_sex(cls, v):
        if v not in ["M", "F"]:
            raise ValueError("Invalid sex")
        return v
```

Referential integrity

If you're working with multiple tables (e.g., patients and visits), you need to ensure foreign keys match.

```
valid_ids = patients['patient_id']
df['id_in_patients'] = df['patient_id'].isin(valid_ids)
```

If you find rows that reference a non-existent patient, that's a red flag.

Constraint enforcement with assertions

You can add assertions at the top of notebooks or scripts:

```
assert df['age'].between(0, 120).all(), "Age out of range
    !"
```

This helps ensure things don't break silently.

Case study: validating a clinical dataset

You receive a dataset with demographics, vital signs, and lab results. You want to ensure:

- Age is 0–120
- Sex is M or F
- All rows have a patient ID
- Glucose values are realistic

Step-by-step:

```
assert df['age'].between(0, 120).all()
assert df['sex'].isin(['M', 'F']).all()
```

```
assert df['patient_id'].notna().all()
assert df['glucose'].between(30, 800).all()
```

Alternatively, you could wrap these into a `pandera` schema.

Logging and documenting rules

Whenever you apply validations, document them. You can:

- Keep a `constraints.md` file describing each rule
- Store validation functions in a module (e.g. `validate.py`)
- Add logging to your pipeline

```
import logging
logging.info("Validated sex field: %d invalid entries",
    (~df['sex'].isin(['M', 'F'])).sum())
```

Summary

- Validation is the practice of enforcing logic and sanity checks on your data.
- Use Python logic, regex, or tools like `pandera` and `pydantic`.
- Catch issues early—don't wait for your model to fail.
- Document all rules applied.

Clean data isn't just about missing values or duplicates—it's also about values that make sense. Validation is your last line of defense before analysis begins.

Logging, Versioning, and Audit Trails

You've cleaned your data. It looks great. But how will someone else know what you changed—and why? And more importantly: how will **you** remember what you did three weeks from now?

That's where logging, versioning, and audit trails come in. These aren't just "nice to have" in regulated environments—they're essential for reproducibility, trust, and scientific integrity.

Why traceability matters

In clinical research, data must comply with ALCOA+ principles:

- **Attributable**: Who made the change?
- **Legible**: Can we read and understand it?
- **Contemporaneous**: Was it recorded at the time?
- **Original**: Do we have the raw version?
- **Accurate**: Is it correct?

The "+" adds: Complete, Consistent, Enduring, and Available.

These principles don't just apply to the data—they apply to your **code**, **decisions**, and **process**.

Keeping raw and cleaned data separate

Never overwrite the original file. Always save cleaned versions with clear names:

```
df.to_csv("cleaned/step3_validated.csv", index=False)
```

Include a README or changelog describing what changed:

```
2025-05-08 | Removed outliers and fixed BMI mismatches.
    Stored in step3_validated.csv
```

You can even log the number of rows affected:

```
removed = initial_len - len(df)
print(f"Removed {removed} rows with invalid age")
```

Logging with the `logging` module

Instead of just printing, log messages properly:

```
import logging
logging.basicConfig(filename='logs/cleaning.log', level=
    logging.INFO)
logging.info("Step 2: Replaced missing BMI values with
    median")
```

Log unexpected values too:

```
bad_sex = df[~df['sex'].isin(['M', 'F'])]
logging.warning(f"Found {len(bad_sex)} invalid sex
    entries")
```

Versioning data with DVC

DVC is like Git for data. You can version `.csv` or `.parquet` files and track them with your code.

Initialize a DVC project:

```
dvc init
dvc add cleaned/step3_validated.csv
git add cleaned/step3_validated.csv.dvc .gitignore
git commit -m "Track validated dataset with DVC"
```

This allows rollback, comparison, and reproducibility.

Recording code versions

Use Git to track your notebooks and scripts. Commit often:

```
git commit -am "Cleaned text fields and recoded
    categories"
```

Tag important stages:

```
git tag -a v1.0 -m "Final cleaned dataset before modeling
    "
```

Jupyter notebooks and cell history

Jupyter keeps execution counts, but it's easy to rerun cells in a different order and lose traceability. Use these tips:

- Run notebooks top to bottom
- Export to HTML or PDF after each milestone
- Use `nbconvert` or tools like `papermill` or `jupytext`

Reproducible scripts

Move cleaning steps into scripts:

```
python clean_data.py
```

This ensures anyone can re-run the pipeline, not just click through a notebook.

Include command-line logging:

```
print("Step 1 complete: dropped duplicates")
```

Or structured logging:

```
import logging
logging.info("Step 3 complete: imputed missing glucose")
```

Case study: versioned cleaning pipeline

You create a script `cleaning_pipeline.py` that:

- Loads raw data from `data/raw.csv`
- Cleans types, missing values, and outliers
- Saves to `cleaned/step_final.csv`
- Logs all actions
- Is tracked by Git and DVC

Now, anyone on your team (or a reviewer) can:

- Re-run the entire process
- See exactly what was changed
- Compare results before and after cleaning

Summary

- Always keep raw and cleaned data separate
- Use logging to document every change
- Version data files with DVC or commit intermediate steps manually
- Track your code with Git and tag important milestones
- Package your cleaning steps into scripts for full reproducibility

Clean data is great. But clean **processes** are even better. That's what makes your work solid, trusted, and ready for the real world.

Case Studies in Data Cleaning

Let's wrap up with some real-world examples. These case studies combine many of the techniques we've explored so far, and show how they're applied to messy, realistic data. Each one starts from a raw dataset and walks through the steps to make it usable.

Case 1: Cleaning Clinical EHR Data

You're given an extract of electronic health records (EHR) with the following columns:

- `patient_id`
- `sex`
- `birth_date`
- `visit_date`
- `height_cm`, `weight_kg`, `bmi`
- `glucose`, `blood_pressure`, `notes`

Problems spotted:

- `sex` values: "M", "F", "male", "FEMALE", "unknown"
- `birth_date` stored as text, some in DD/MM/YYYY, some in MM-DD-YYYY
- `bmi` doesn't match height and weight in many cases
- Notes contain typos and irregular whitespace

Cleaning steps:

```
# Standardize sex
sex_map = {'male': 'M', 'female': 'F', 'f': 'F', 'm': 'M'}
df['sex'] = df['sex'].str.strip().str.lower().replace(sex_map)
df['sex'] = df['sex'].where(df['sex'].isin(['M', 'F']), pd.NA)

# Parse birth_date
from dateutil import parser

def try_parse(date_str):
    try:
        return parser.parse(date_str, dayfirst=True)
    except:
        return pd.NaT

df['birth_date'] = df['birth_date'].apply(try_parse)

# Recalculate BMI
height_m = df['height_cm'] / 100
df['bmi_calc'] = df['weight_kg'] / (height_m ** 2)
df['bmi_diff'] = (df['bmi'] - df['bmi_calc']).abs()
df['bmi_flag'] = df['bmi_diff'] > 1.5

# Clean notes
import re
df['notes'] = df['notes'].str.strip().str.lower().str.replace(r'\s+', ' ', regex=True)
```

Results:

- Cleaned categorical fields
- Consistent date format
- Mismatched BMI flagged for review

- Notes simplified for NLP or grouping

Case 2: Public Health Indicators (WHO)

You're working with a dataset of country-level health metrics. It includes:

- `country`, `year`
- `life_expectancy`, `population`, `gdp_per_capita`
- `urban_pct`, `mortality_rate`, `vaccination_coverage`

Problems spotted:

- Missing `gdp_per_capita` in some countries
- Duplicate rows for some (country, year) pairs
- `urban_pct` has values over 100
- `vaccination_coverage` includes "n/a"

Cleaning steps:

```
# Drop duplicates
df = df.drop_duplicates(subset=['country', 'year'])

# Convert to numeric
df['gdp_per_capita'] = pd.to_numeric(df['gdp_per_capita'
    ], errors='coerce')
df['vaccination_coverage'] = pd.to_numeric(df['
    vaccination_coverage'], errors='coerce')

# Cap invalid values
df['urban_pct'] = df['urban_pct'].clip(upper=100)
```

```
10
11  # Impute GDP using country median
12  df['gdp_per_capita'] = df.groupby('country')['
        gdp_per_capita'].transform(
13      lambda x: x.fillna(x.median())
14  )
```

Results:

- Cleaned dataset ready for time series analysis
- No impossible values (e.g., 120% urban population)
- Reproducible script with clear assumptions

Case 3: LC-MS Metabolomics Data

You're analyzing untargeted LC-MS metabolomics data with compound intensities across multiple samples:

- `sample_id`, `compound_name`, `retention_time`, `area`, `flag`

Issues found:

- Some compounds have duplicate peaks
- `retention_time` varies slightly between replicates
- Some flags are "bb", "MM", or missing
- Intensities vary by 10x between replicates

Cleaning actions:

```
# Filter based on flag
valid_flags = ['']  # empty string means no issues
df = df[df['flag'].isin(valid_flags)]

# Group by sample and compound
summary = df.groupby(['sample_id', 'compound_name']).agg
    ({
    'area': 'mean',
    'retention_time': 'median'
}).reset_index()

# Log number of removed flagged rows
removed = len(df_original) - len(df)
print(f"Removed {removed} rows based on flag status")
```

Outcome:

- Aggregated dataset with clean peaks per sample
- Median retention time per compound
- Flags documented and discarded safely

Summary

These case studies show that real data is messy in many ways—text, dates, numbers, duplicates, logic, and structure. Cleaning it well takes patience, creativity, and rigor. But once you do, you unlock insights you can trust—and models that work.

You're not just fixing a spreadsheet. You're preparing evidence.

References and Resources

A well-cleaned dataset is a silent achievement—it often goes unnoticed, but it makes everything else possible. Below are key references, tools, and datasets mentioned throughout the book, especially in the case studies.

Key References

- Karr, A. F., Sanil, A. P., & Banks, D. L. (2006). *Data quality: A statistical perspective*. Statistical Methodology, 3(2), 137–173. https://doi.org/10.1016/j.stamet.2005.08.005
- EMA (2010). *Reflection paper on expectations for electronic source data and data transcribed to electronic data collection tools in clinical trials*. European Medicines Agency. https://www.ema.europa.eu/en/documents/scientific-guideline/reflection-paper-expectations-electronic-source-data-data-transcribed-electronic-data-collection_en.pdf
- FDA (2018). *Data Integrity and Compliance With Drug CGMP: Questions and Answers*. https://www.fda.gov/media/119267/download
- Pandera documentation: https://pandera.readthedocs.io
- Pydantic documentation: https://docs.pydantic.dev/
- DVC (Data Version Control): https://dvc.org
- Missingno: https://github.com/ResidentMario/missingno

Datasets Used in Case Studies

Clinical EHR Dataset (Synthetic Example)

To replicate the clinical cleaning steps, you can use a synthetic but realistic dataset like:

- Synthea open EHR data — Fully synthetic patient records, downloadable in CSV format.
- Sample dataset used in SeerPy (cleaned EHR structure): https://github.com/SEERData/SeerPy (exploration purposes)

WHO Health Indicators Dataset

- WHO Global Health Observatory Data Repository
- Specific indicators like life expectancy, mortality, and immunization: https://www.who.int/data/gho/indicator-metadata-registry

Metabolomics LC-MS Dataset

- Example LC-MS data (MTBLS files):
  - MetaboLights - MTBLS135 — Plasma metabolome data.
  - MTBLS404 - Human serum NMR and MS data

These provide compound tables with retention time, area, and sample annotations.

Tools Used Throughout

Tool	Purpose	Link
pandas	Data manipulation	https://pandas.pydata.org
seaborn/matplotlib	Visualization	https://seaborn.pydata.org
scikit-learn	Imputation, scaling, model prep	https://scikit-learn.org
pandera	DataFrame schema validation	https://pandera.readthedocs.io
pydantic	Row-level validation and models	https://docs.pydantic.dev
DVC	Data versioning and reproducibility	https://dvc.org
missingno	Visualizing missing data	https://github.com/ResidentMario/missingno

---

If you want to go further, consider browsing the datasets on:

- Kaggle Datasets
- UCI Machine Learning Repository
- OpenML

And remember: behind every trusted insight is a dataset that someone took the time to clean properly. Now, that someone is you.


rubió
Metabolomics