WEEK 3

Data Cleaning II + Real Dataset

Production-ready cleaning with outliers, strings, dates, and full pipelines

Day 11	**Outlier Strategies**	Capping, flooring, removal, model sensitivity
Day 12	**String & Date Cleaning**	Text normalization, datetime parsing, time zones
Day 13	**Large Dataset Cleaning**	Chunking, sampling, vectorization, performance
Day 14	**Full Cleaning Pipeline**	Modular design, order of operations, validation
Day 15	**Mini Project**	End-to-end cleaning of real messy dataset

BASE DATA SETUP

```
import numpy as np, pandas as pd, matplotlib.pyplot as plt, seaborn as sns
np.random.seed(42)
# Simulate heavy-tailed income with extreme outliers
```

DAY 11

Outlier Strategies

Capping, Flooring, Removal, and Model Impact

OBJECTIVES

- Review outliers and how they arise
- Implement capping (winsorization) and flooring
- Use percentiles and IQR for thresholds
- Visualize before/after with boxplots
- Reason about when NOT to remove outliers

ACTIVITY

Implement 99th percentile capping function.

Apply to real feature and visualize before/after.

ASSESSMENT

Notebook with code, boxplots, and written justification for chosen strategy.

DAY 11 · OUTLIER STRATEGIES

Outliers in Real Datasets — Brief Review

Not automatically bad data - may be critical signals

Outlier Sources

- True extreme values (startup exit, large hospital bill)
- Data-entry mistakes (extra zero in salary)
- System artifacts (duplicate aggregation, unit conversion errors)

Why 'Just Remove Big Values' Fails

- Extremes may be most informative (fraud, default risk)
- Removal shifts means, variances, decision boundaries
- Outliers may represent specific subgroups → unfairness

VISUALIZING WITH BOXPLOT

```
income = np.concatenate([np.random.lognormal(10, 0.5, 5000), [1e7, 2e7, 5e7]])
df11 = pd.DataFrame({"income": income})
plt.figure(figsize=(6,4)); sns.boxplot(x=df11["income"])
plt.xscale("log"); plt.title("Income Distribution with Outliers")
```

DAY 11 · OUTLIER STRATEGIES

Capping Outliers (Winsorization)

Replace extreme values with percentile thresholds

Real-World Motivation

- Extremely large values distort loss functions (squared error)
- Rather than discard, cap to limit leverage
- Keeps all records but reduces extreme influence

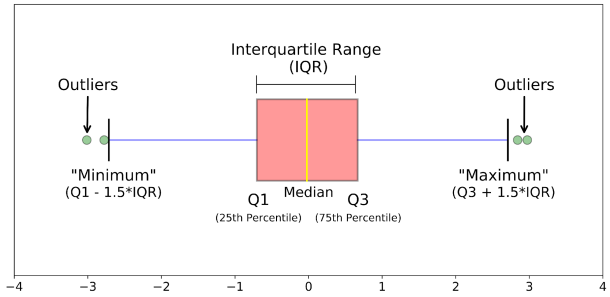
When Capping Is Reasonable

- Tails are real but shouldn't dominate
- Goal is robust prediction, not precise tail estimation
- Sample size is limited

WINSORIZATION AT 1ST AND 99TH PERCENTILES

```
def winsorize_series(s: pd.Series, lower_q=0.01, upper_q=0.99) -> pd.Series:
    lower, upper = s.quantile(lower_q), s.quantile(upper_q)
    return s.clip(lower=lower, upper=upper)

df11["income_cap_1_99"] = winsorize_series(df11["income"], 0.01, 0.99)
```



DAY 11 · OUTLIER STRATEGIES

Flooring and Asymmetric Capping

Different thresholds for lower and upper bounds

Use Cases

- Negative values impossible by domain (negative counts)
- Very small positives = sensor noise, not real signal
- Asymmetric distributions where only one tail problematic

Asymmetric Strategy

Different quantiles for lower/upper (e.g., 5th percentile floor, 99.5th cap)

Risk: aggressive flooring may obscure economically vulnerable groups

ASYMMETRIC CAPPING WITH .clip()

```
def asymmetric_cap(s: pd.Series, lower_q=0.05, upper_q=0.995) -> pd.Series:
    lower, upper = s.quantile(lower_q), s.quantile(upper_q)
    return s.clip(lower=lower, upper=upper)

df11["income_cap_asym"] = asymmetric_cap(df11["income"], 0.05, 0.995)
```

DAY 11 · OUTLIER STRATEGIES

Removal Strategy — When Dropping Is Appropriate

Most aggressive - eliminates extreme rows entirely

Appropriate Scenarios

- Confirmed data-entry errors (salary 1B when max is 1M)
- Obvious system artifacts (duplicated aggregation, corrupted records)

Inappropriate Scenarios

- Rare but legitimate high-impact events (ICU stays, large purchases)
- Small datasets where every row matters
- Risk: model blind to critical regimes

REMOVAL USING PERCENTILE MASK

```
upper_99 = df11["income"].quantile(0.99)
mask_keep = df11["income"] <= upper_99
df11_removed = df11[mask_keep].copy()

print("Original rows:", len(df11), "After removal:", len(df11_removed))
```

Visual Comparison of Capping vs Removal

Boxplots reveal shifts in quartiles and tails

What to Look For

- How far whiskers extend after treatment
- Whether central bulk (IQR) is significantly altered
- Whether many points cluster at cap boundary

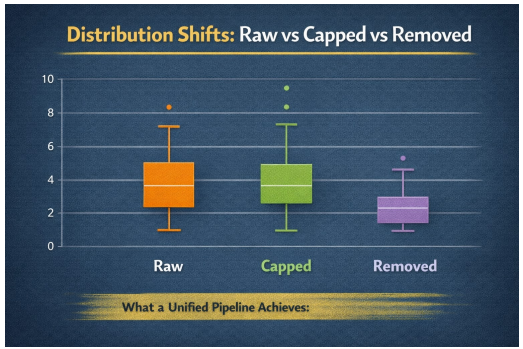
Interpretation

- Raw: long whiskers, extreme outliers
- Capped: whiskers truncated, same N
- Removed: shorter whiskers, reduced N

BOXPLOTS FOR RAW, CAPPED, AND REMOVED

```
data_to_plot = [df11["income"], df11["income_cap_1_99"], df11_removed["income"]]
labels = ["Raw", "Capped (1-99%)", "Removed (>99%)"]

plt.figure(figsize=(8,4)); sns.boxplot(data=data_to_plot)
plt.xticks(range(len(labels)), labels); plt.xscale("log")
```



DAY 11 · OUTLIER STRATEGIES

ML Model Sensitivity to Outlier Strategies

Different models respond differently to extreme values

Linear/Logistic

Highly sensitive due to squared/log loss

Outliers → large gradients, unstable parameters

Tree-Based

More robust but still affected

Splits may focus on rare extremes

Regularization

L1/L2 partially mitigates but doesn't fix underlying data

LOGISTIC REGRESSION WITH/WITHOUT CAPPING

```
from sklearn.linear_model import LogisticRegression
threshold = df_model["income"].quantile(0.9)
df_model["target"] = (df_model["income"] >= threshold).astype(int)

clf_raw = LogisticRegression().fit(X_raw, y); clf_cap = LogisticRegression().fit(X_cap, y)
print("Coef raw:", clf_raw.coef_[0][0], "Coef capped:", clf_cap.coef_[0][0])
```

DAY 11 · OUTLIER STRATEGIES

When NOT to Remove or Cap Outliers

Sometimes extremes ARE the signal

Domains Where Extremes Matter

- Fraud detection: rare high-value fraudulent transactions
- Risk modeling: catastrophic failures, defaults
- Medical outcomes: rare severe complications

Alternative Strategies

- Use robust loss functions (Huber loss)
- Model tails separately (two-stage modeling)
- Weight rare events appropriately

IDENTIFYING TAIL FOR SPECIAL TREATMENT

```
high_tail = df11["income"] >= df11["income"].quantile(0.995)
df_tail = df11[high_tail]
df_main = df11[~high_tail]
print("Tail count:", len(df_tail), "Main count:", len(df_main))
```

DAY 11 · OUTLIER STRATEGIES

Implementing a 99th Percentile Capping Function

Reusable function for production pipelines

Requirements

- Percentile thresholds computed from data
- Ability to cap only upper tail, or both
- Clear documentation of chosen percentiles
- Return thresholds for logging/reproducibility

Extensions

- Add lower_q parameter for two-sided capping
- Return both capped series and thresholds
- Support column-wise application

FUNCTION THAT CAPS AT 99TH PERCENTILE

```
def cap_at_percentile(s: pd.Series, upper_q: float = 0.99) -> pd.Series:
    """Cap values in a numeric series at a given upper percentile."""
    upper = s.quantile(upper_q)
    return s.clip(upper=upper)

df11["income_cap_99"] = cap_at_percentile(df11["income"], upper_q=0.99)
```

DAY 11 · ACTIVITY & ASSESSMENT

Outlier Capping Practice

Implement and visualize capping strategies

Activity Steps

1. Load real numeric column (e.g., transaction amount)
2. Plot boxplot and histogram (consider log-scale)
3. Implement cap_at_percentile at 99th
4. Apply to feature, create capped column
5. Plot before/after boxplots on same scale
6. Compare summary stats (mean, median, std)

Assessment

Conceptual:

- Difference between winsorization and removal?
- When is 99th percentile harmful vs helpful?

Code: Implement cap_outliers(df, column, upper_q) that prints threshold and fraction capped.

SKELETON

```
df_real["amount_capped"] = cap_at_percentile(df_real["amount"], 0.99)
fig, axes = plt.subplots(1, 2); sns.boxplot(x=df_real["amount"], ax=axes[0])
sns.boxplot(x=df_real["amount_capped"], ax=axes[1])
```

DAY 12

String & Date Cleaning

Text Normalization and Datetime Handling

OBJECTIVES

- Clean and normalize user-entered text
- Standardize categories to canonical forms
- Parse heterogeneous date formats
- Localize naive datetimes to time zones
- Understand pitfalls of naive datetime handling

ACTIVITY

Clean and standardize city column and timestamps.

ASSESSMENT

Code with cleaned text/dates, plus written explanation of decisions.

DAY 12 • STRING & DATE CLEANING

Messy User-Entered Text

Whitespace, case, and noise create spurious categories

Typical Problems

- Leading/trailing whitespace: ' new york' vs 'new york '
- Inconsistent case: 'NEW YORK' vs 'new york'
- Embedded punctuation: 'san-francisco', 'San Francisco!'
- Abbreviations: 'nyc', 'NY', 'N.Y.'

Why Basic Cleaning Is Insufficient

- Case normalization doesn't resolve synonyms
- Naive replacement rules can merge distinct categories
- Without explicit mapping, model sees many rare categories

BASIC WHITESPACE AND CASE NORMALIZATION

```
df12 = pd.DataFrame({"raw_city": [" new york", "New york ", "NEW YORK", "nyc", "NYC"]})
df12["city_clean_basic"] = (
    df12["raw_city"].str.strip().str.lower()  # Remove whitespace, lowercase
)
print(df12[["raw_city", "city_clean_basic"]])
```

DAY 12 • STRING & DATE CLEANING

Removing Special Characters and Normalizing Separators

Hyphens and punctuation create artificial categories

Problems

- 'san-francisco' vs 'San Francisco' = distinct values
- 'London!' pollutes category space

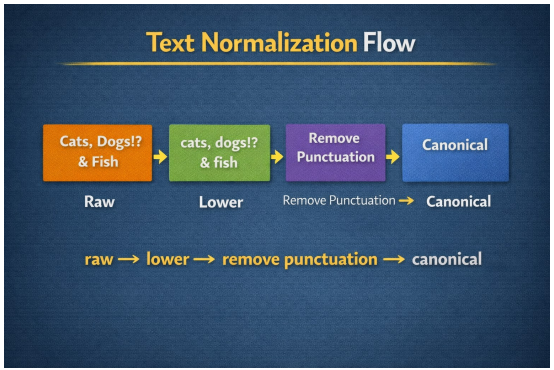
Strategy: Replace separators with spaces, remove non-letters

Risks

- Over-aggressive removal (diacritics, non-Latin) can erase meaningful distinctions
- Always validate against domain requirements

NORMALIZING WITH .str AND REGEX

```
df12["city_clean_sep"] = (
    df12["city_clean_basic"]
    .str.replace("-", " ", regex=False)              # Hyphens to spaces
    .str.replace(r"[^a-z\s]", "", regex=True)       # Keep only letters and spaces
    .str.replace(r"\s+", " ", regex=True).str.strip() )  # Collapse multiple spaces
```



DAY 12 • STRING & DATE CLEANING

Standardizing Categories — Mapping to Canonical Values

Explicit mapping for synonyms and abbreviations

Example Mapping

- 'new york', 'nyc', 'ny' → 'new york'
- 'san francisco', 'sanfrancisco' → 'san francisco'

Without mapping: sparse one-hot, unstable models

Risks

- Incorrect mappings (e.g., 'ny' for 'New Year' code)
- Must validate against domain knowledge
- Document all mappings for auditability

MAPPING USING .replace OR .map

```
canonical_map = {"new york": "new york", "nyc": "new york", "ny": "new york",
                 "san francisco": "san francisco", "sanfrancisco": "san francisco"}

df12["city_token"] = df12["city_clean_sep"].str.replace(" ", "", regex=False)
df12["city_canonical"] = df12["city_token"].map(canonical_map).fillna(df12["city_clean_sep"])
```

DAY 12 • STRING & DATE CLEANING

Heterogeneous Date Formats — Parsing to Datetime

Real datasets have multiple date formats

Common Issues

- Mix of ISO ('2024-01-01'), slashed ('01/01/2024')
- Day-first vs month-first ambiguity ('04/05/2024')
- Mixed separators (-, /, T)
- Timestamps with varying precision

Strategy

- Use pd.to_datetime(errors='coerce')
- Inspect NaT count after parsing
- Use dayfirst parameter where appropriate

INITIAL PARSING ATTEMPT

```
df12["raw_signup"] = ["2024-01-01 10:00", "01/01/2024 15:00", "2024/01/01"]
df12["signup_dt_raw"] = pd.to_datetime(df12["raw_signup"], errors="coerce", infer_datetime_format=True)

print(df12[["raw_signup", "signup_dt_raw"]])
print("NaT count:", df12["signup_dt_raw"].isna().sum())
```

DAY 12 • STRING & DATE CLEANING

Handling Ambiguous Date Formats and Locale Issues

'04/05/2024' may mean April 5 or May 4

Strategy

- If source locale known, use dayfirst or format string
- Otherwise, mark ambiguous as missing
- Separate US-style and ISO-like dates with masks

Risks

- Guessing locale incorrectly shifts dates by months
- Silently breaks seasonality features
- Can corrupt model calibration entirely

SEPARATING US-STYLE AND ISO DATES

```
us_mask = df12["raw_signup"].str.contains(r"\d{2}/\d{2}/\d{4}", regex=True)
iso_mask = ~us_mask

df12.loc[iso_mask, "signup_dt"] = pd.to_datetime(df12.loc[iso_mask, "raw_signup"], errors="coerce")
df12.loc[us_mask, "signup_dt"] = pd.to_datetime(df12.loc[us_mask, "raw_signup"], errors="coerce",
dayfirst=False)
```

DAY 12 • STRING & DATE CLEANING

Time Zones, Localization, and Conversion

Naive vs aware datetimes for cross-region systems

Key Concepts

- Naive datetime: no time zone (tzinfo=None)
- Aware datetime: timezone-aware (tzinfo set)
- tz_localize: attach timezone to naive (interpretation)
- tz_convert: convert between zones (transformation)

Pitfall

Using naive timestamps from different regions as if same zone → incorrect comparisons and aggregations

LOCALIZING TO US/EASTERN AND CONVERTING TO UTC

```
df12["signup_dt_local"] = df12["signup_dt"].dt.tz_localize("US/Eastern", ambiguous="NaT",
nonexistent="NaT")
df12["signup_dt_utc"] = df12["signup_dt_local"].dt.tz_convert("UTC")
```

DAY 12 · STRING & DATE CLEANING

Pitfalls of Naive Datetime Handling in ML

Silent errors in time-based features

Typical Mistakes

- Comparing timestamps from different zones directly
- Aggregating by hour without common reference
- Ignoring daylight saving transitions → duplicate/missing hours

ML Impact

- Misaligned features confuse models about daily/weekly cycles
- Fairness concerns if time zones correlate with demographics

DERIVING TIME-BASED FEATURES AFTER LOCALIZATION

```
df12["signup_hour_local"] = df12["signup_dt_local"].dt.hour
df12["signup_weekday_local"] = df12["signup_dt_local"].dt.dayofweek

print(df12[["signup_dt_local", "signup_hour_local", "signup_weekday_local"]].head())
```

DAY 12 · ACTIVITY & ASSESSMENT

String & Date Cleaning Practice

Clean city and datetime columns end-to-end

Activity Steps

1. Load dataset with city names and timestamps
2. Strip whitespace, normalize case
3. Remove/normalize separators and special chars
4. Design variant → canonical mapping
5. Parse dates with errors='coerce'
6. Localize to relevant timezone, convert to UTC
7. Create derived features (hour, day of week)

Assessment

Conceptual:

- Why can naive datetime parsing lead to biased model estimates?
- How might time zones interact with seasonality features?

Code: Implement standardize_city() and localize_and_convert()

SKELETON

```
df_users["city_clean"] = df_users["city"].str.strip().str.lower().str.replace("-", " ", regex=False)
df_users["signup_dt"] = pd.to_datetime(df_users["signup_time"], errors="coerce")
df_users["signup_dt_utc"] = df_users["signup_dt"].dt.tz_localize("US/Eastern").dt.tz_convert("UTC")
```

DAY 13

Large Dataset Cleaning

Chunking, Sampling, and Performance Optimization

OBJECTIVES

- Define 'large' in practical settings
- Use chunked reading (chunksize) for memory-constrained data
- Sample for iterative cleaning design
- Compare vectorized vs .apply() vs loops
- Measure and identify bottlenecks

ACTIVITY

Work with large dataset to identify and optimize slow steps.

ASSESSMENT

Written strategies for large datasets plus optimization code.

DAY 13 • LARGE DATASETS

What Makes a Dataset 'Large' in Practice

Relative to available memory and latency requirements

Practical Criteria

- Dataset size approaches/exceeds available RAM
- End-to-end cleaning takes too long interactively
- Repeated reloads for experimentation are slow

Why Basic Cleaning Is Insufficient

- Many chained .assign() calls = expensive copies
- Python loops over rows (df.iterrows()) become prohibitive
- Must think about memory and compute together

ROUGH MEMORY ESTIMATE

```
import os
file_path = "large_users.csv"
size_bytes = os.path.getsize(file_path)
size_mb = size_bytes / (1024 * 1024)
print(f"File size: {size_mb:.2f} MB")  # First indication of feasibility
```

DAY 13 • LARGE DATASETS

Reading Data in Chunks with chunksize

Process large files in manageable pieces

Benefits

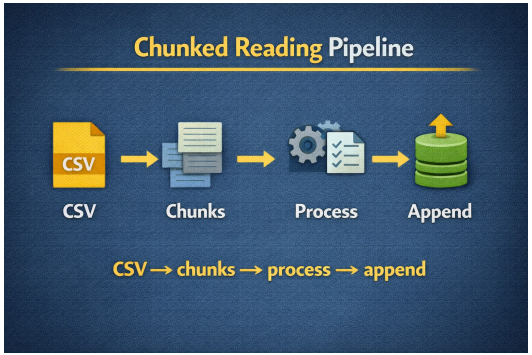
- Memory usage bounded by chunk size
- Stream through data progressively
- Can compute aggregates incrementally

Trade-offs

- More complex control flow (manual concatenation)
- Some operations need coordination across chunks (global deduplication, percentiles)

BASIC CHUNKED ITERATION

```
chunks = pd.read_csv("large_users.csv", chunksize=50_000)
total_rows, sum_income = 0, 0.0
for chunk in chunks:
    total_rows += len(chunk)
    sum_income += chunk["income"].sum()
mean_income = sum_income / total_rows
```



DAY 13 • LARGE DATASETS

Sampling for Iterative Cleaning Design

Iterate quickly on sample, then scale

Strategy

- Load random subset (e.g., 50,000 rows) for EDA
- Verify patterns hold across full dataset via per-chunk checks
- Only run full pipeline after logic is validated

Risks

- Non-random sampling (first N rows) may miss patterns
- Rare issues (categories, outliers) might not appear in small samples

RANDOM ROW SAMPLING DURING LOAD

```
sample_df = pd.read_csv("large_users.csv", nrows=50_000)

print(sample_df.head())
print(sample_df["city"].value_counts())  # Quick check of category distribution
```

DAY 13 • LARGE DATASETS

Vectorization vs .apply() vs Explicit Loops

Understanding performance hierarchy

Performance Hierarchy

Best: Pure vectorized (df['x'] * 2, df['x'].clip())

Medium: .apply() with simple functions

Slowest: Row-wise .apply(axis=1) or for loops

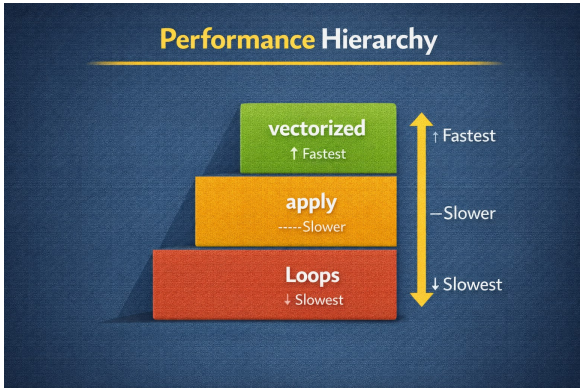
ML Impact

Efficient cleaning → more experiments → better models

Inefficient cleaning may limit experimentation entirely

TIMING COMPARISON

```
%timeit sample["income_scaled_vec"] = sample["income"] / 1000.0       # Fast
%timeit sample["income_scaled_apply"] = sample["income"].apply(lambda x: x/1000)  # Slower
```



DAY 13 • LARGE DATASETS

Avoiding Python Loops in Cleaning

Express transformations as column-wise operations

ANTI-PATTERN (AVOID)

```
clean_values = []
for _, row in sample_df.iterrows():
    clean_values.append(row["income"] / 1000.0)
sample_df["income_scaled_loop"] = clean_values
```

RECOMMENDED PATTERN

```
# Column-wise operation
sample_df["income_scaled"] =
sample_df["income"] / 1000.0

# Conditional without loops
sample_df["high_flag"] = (sample_df["income"] >
threshold).astype(int)
```

Key Insight

Use .where, .mask, .clip, .fillna for conditional logic instead of row-by-row iteration. Keeps pipeline scalable and maintainable.

DAY 13 • LARGE DATASETS

Measuring Performance and Identifying Bottlenecks

Without measurement, optimization is guesswork

Approaches

- %timeit in interactive environments
- time.perf_counter() in scripts
- Profile memory via system tools or specialized libraries

Use Performance Data

- Guides whether to refactor or optimize
- Identifies if distributed solutions needed
- Prioritizes effort on actual bottlenecks

TIMING A CLEANING FUNCTION

```
import time
def clean_chunk(chunk): return chunk.copy()

start = time.perf_counter()
cleaned = clean_chunk(sample_df)
elapsed = time.perf_counter() - start
print(f"Chunk cleaned in {elapsed:.3f} seconds")
```

DAY 13 · ACTIVITY & ASSESSMENT

Large Dataset Practice

Chunked cleaning with performance measurement

Activity Steps

1. Get CSV with 100K+ rows
2. Estimate file size, consider RAM
3. Design chunked cleaning loop
4. Apply cleaning to each chunk
5. Write cleaned chunks to new file
6. Measure runtime and memory
7. Compare with full-file approach

Assessment

Short answer:

- Three strategies for datasets that don't fit in RAM?
- How does vectorization help performance?

Code: Implement process_large_file(path_in, path_out, chunksize)

CHUNKED CLEANING SKELETON

```
reader = pd.read_csv("large_users.csv", chunksize=50_000)
for i, chunk in enumerate(reader):
    cleaned = clean_chunk(chunk)
    mode, header = ("w", True) if i == 0 else ("a", False)
    cleaned.to_csv("cleaned.csv", mode=mode, header=header, index=False)
```

DAY 14

Full Cleaning Pipeline

Modular Design, Order of Operations, Validation

OBJECTIVES

- Understand why pipelines are essential for ML
- Design function-based modular components
- Establish clear order of operations
- Integrate logging and validation
- Plan for pipeline failure points

ACTIVITY

Build and test a master clean_data() function.

ASSESSMENT

Code review focusing on clarity, documentation, reproducibility.

DAY 14 · CLEANING PIPELINE

Why Cleaning Pipelines Matter for ML

Ad hoc cleaning in notebooks is fragile

Benefits of Pipelines

- Reproducibility: same raw data → same cleaned data
- Debuggability: trace problems to specific steps
- Deployment: same logic in batch and online systems

Problems with Ad Hoc Cleaning

- Hidden state across notebook cells
- Hard to re-run exactly after changes
- Difficult for team to understand/trust

SIMPLE PIPELINE SHAPE

```
def clean_data(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df = clean_types(df)
    df = clean_missing(df)
    df = handle_outliers(df)
    df = clean_strings_and_dates(df)
    validate_cleaned(df)
    return df
```

DAY 14 · CLEANING PIPELINE

Function-Based Cleaning and Modular Design

Single responsibility per function promotes testability

Design Principles

- Single responsibility per function
- Clear input/output contracts (DataFrame in/out)
- No hidden global state
- Each function independently testable

Each function is reusable across projects and easily unit-tested.

EXAMPLE HELPER FUNCTIONS

```
def clean_types(df):
    out = df.copy()
    out["age"] = pd.to_numeric(out["age"],
errors="coerce")
    out["income"] =
pd.to_numeric(out["income"], errors="coerce")
    return out

def clean_missing(df):
    out = df.copy()
    out["age"] =
out["age"].fillna(out["age"].median())
    return out
```

DAY 14 · CLEANING PIPELINE

Order of Operations and Pipeline Stability

Transformation order has major implications

Recommended Order

1. Type normalization (numeric, categorical, datetime)
2. Missing-value handling (imputation, indicators)
3. Outlier handling (capping, transformation)
4. String and category cleaning
5. Date/time feature extraction

Why Order Matters

- Outlier detection on wrong types = meaningless
- Imputation before type conversion hides errors
- Date features from misparsed timestamps = wrong

Change order deliberately and document!

INTEGRATING ORDER INTO clean_data()

```
def clean_data(df):
    df = clean_types(df)           # 1. Types first
    df = clean_missing(df)         # 2. Then missing values
    df = handle_outliers(df)       # 3. Then outliers
    df = clean_strings_and_dates(df)  # 4. Then strings/dates
    df = add_features(df)          # 5. Finally derived features
    return df
```

DAY 14 · CLEANING PIPELINE

Logging and Basic Validation

Catch anomalies early, ensure expected behavior

Logging

- Record row counts, missingness summary
- Log outlier thresholds used
- Store logs with pipeline runs for auditability

Validation

- Assert key invariants (no negative ages)
- Check ranges and category memberships
- Verify required columns present

LOGGING/VALIDATION PATTERNS

```
def validate_cleaned(df):
    assert df["age"].min() >= 0, "Negative ages found"
    assert df["income"].notna().all(), "Income still has NaN"

def log_summary(df, step_name):
    print(f"--- {step_name} ---\nRows: {len(df)}\nMissing age: {df['age'].isna().sum()}")
```

DAY 14 · CLEANING PIPELINE

Pipeline Failure Points and Error Handling

Anticipate and handle unexpected data gracefully

Common Failure Points

- Unexpected types (all strings where numbers expected)
- New categories not covered by mapping
- Invalid dates (impossible calendar dates)
- Schema changes from upstream systems

Error-Handling Strategies

- Fail fast with clear error messages
- Route problematic records to quarantine table
- Log frequencies of errors for prioritization

SAFE TYPE CONVERSION WITH ERROR LOGGING

```
def safe_to_numeric(s, col_name):
    converted = pd.to_numeric(s, errors="coerce")
    n_invalid = converted.isna().sum() - s.isna().sum()
    if n_invalid > 0: print(f"[WARN] {n_invalid} invalid in {col_name}")
    return converted
```

DAY 14 • ACTIVITY & ASSESSMENT

Pipeline Building Practice

Build master clean_data() function

Activity Steps

1. Choose realistic raw dataset
2. Design clean_data() pipeline that:
   - Normalizes types
   - Handles missing with indicators
   - Caps/transforms outliers
   - Cleans text and parses dates
3. Integrate logging and validation
4. Run end-to-end and inspect results

Assessment

Review questions:

- How verify pipeline is deterministic?
- How adapt for daily batch processing?
- How unit-test individual functions?

Submit pipeline code + written rationale.

SKELETON

```
def clean_strings_and_dates(df):
    out = df.copy()
    out["city"] = out["city"].str.strip().str.lower()
    out["signup_time"] = pd.to_datetime(out["signup_time"], errors="coerce").dt.tz_localize("UTC")
    return out
```

DAY 15

Mini Project

Clean a Real Dataset End-to-End

OBJECTIVES

- Apply all Week 3 techniques in realistic context
- Document cleaning decisions and rationale
- Produce analysis-ready dataset
- Make realistic trade-offs (drop vs impute, cap vs transform)

ACTIVITY

Clean raw dataset end-to-end, from ingestion to final feature table.

ASSESSMENT

GitHub submission with code, cleaned data, and documentation.

DAY 15 · MINI PROJECT

Selecting and Inspecting a Real Dataset

First step: understand structure and problems

Dataset Requirements

- Numeric variables likely to have outliers
- Text/categorical fields (cities, product names)
- Date/time fields (ideally with timezone considerations)
- Sufficient size to make performance non-trivial

Initial Inspection

- Load small sample to inspect columns, dtypes
- Identify likely targets for outlier strategies
- Identify string cleaning and date handling needs

INITIAL SAMPLING AND INSPECTION

```
df_raw = pd.read_csv("real_dataset.csv")
print(df_raw.head())
print(df_raw.info())
print(df_raw.nunique().sort_values(ascending=False).head(10))
```

DAY 15 · MINI PROJECT

Designing an End-to-End Cleaning Plan

Outline steps before writing code

Plan Components

- Identify key target and predictor variables
- Decide type conversions and expected dtypes
- Specify missing-value strategies per column
- Decide outlier strategies (percentile capping)
- Outline text and date cleaning requirements

ML Considerations

- Which features matter most for modeling?
- Clean critical features most carefully
- Consider fairness/bias for demographic fields

PLAN AS METADATA DICTIONARY

```
cleaning_plan = {
    "age": {"type": "float", "missing": "median_imp", "outliers": "cap_99"},
    "income": {"type": "float", "missing": "median_imp", "outliers": "log1p_cap_99"},
    "city": {"type": "category", "clean": "canonical_city"},
    "signup_time": {"type": "datetime", "tz": "UTC"}
}
```

DAY 15 · MINI PROJECT

Implementing the Project Cleaning Pipeline

Core function: raw data → cleaned DataFrame

Requirements

- Incorporate type conversion, missing values, outliers, strings, dates
- Use vectorized operations where possible
- Include logging summarizing key changes

Starting point - refine with dataset-specific logic.

PROJECT-LEVEL clean_data

```
def clean_data_project(df_raw):
    df = df_raw.copy()
    # Types
    df["age"] = pd.to_numeric(df["age"], errors="coerce")
    df["income"] = pd.to_numeric(df["income"],
errors="coerce")
    df["signup_time"] = pd.to_datetime(df["signup_time"],
errors="coerce")
    # Missing
    df["age_missing"] = df["age"].isna().astype(int)
    df["age"] = df["age"].fillna(df["age"].median())
    df["income_missing"] = df["income"].isna().astype(int)
    df["income"] = df["income"].fillna(df["income"].median())
    # Outliers
    df["income"] =
df["income"].clip(upper=df["income"].quantile(0.99))
    # Strings and dates
    df["city"] = df["city"].str.strip().str.lower()
    df["signup_time"] =
df["signup_time"].dt.tz_localize("UTC")
    return df
```

DAY 15 · MINI PROJECT

Documenting Cleaning Decisions and Rationale

Explain not just what, but why

Key Elements

For each transformation:

- What problem it addresses
- Why you chose that strategy over alternatives
- Potential risks or downsides
- Assumptions about data-generating process

ML Impact

Explain how cleaning choices expected to affect:

- Model performance
- Fairness and bias
- Generalization to new data

DECISION LOG STRUCTURE

```
cleaning_decisions = {
    "income_cap_99": "Cap income at 99th percentile to reduce influence of extreme values while keeping
all records.",
    "age_median_imp": "Impute missing age with global median; less sensitive to outliers than mean."
}
```

DAY 15 · MINI PROJECT

Verifying Correctness and Producing Analysis-Ready Dataset

Validate before using for modeling

Verification Steps

- Check dtypes of all key columns
- Inspect summary stats (min, max, mean, percentiles)
- Inspect category distributions for text fields
- Verify datetimes are timezone-aware
- Confirm no unexpected NaN in critical columns

When Checks Pass

Dataset is ready for downstream analysis or modeling

Save cleaned version (CSV or Parquet) for reproducibility

BASIC VERIFICATION

```
df_clean = clean_data_project(df_raw)
print(df_clean.info())
print(df_clean[["age", "income"]].describe())
print(df_clean["city"].value_counts().head())
print(df_clean["signup_time"].dt.tz)
```

DAY 15 · MINI PROJECT

Full Dataset Cleaning Submission

Integrate all Week 3 techniques

Activity Instructions

1. Select real dataset (public or provided)
2. Design and implement clean_data_project()
3. Run pipeline, generate cleaned dataset
4. Save as CSV or Parquet
5. Prepare documentation:
   - Dataset and its issues
   - All cleaning steps with rationales
   - Trade-offs and limitations

Submission Requirements

GitHub repository containing:

- Raw dataset or pointer to it
- Cleaning code with clean_data_project()
- Cleaned dataset file
- README/report explaining decisions and risks

This integrates all Week 3 into production workflow.

Week 3 Complete!

Data Cleaning II — Production-Ready Skills

Outlier Strategies	Capping (winsorization), flooring, removal; model sensitivity; when NOT to remove
String & Date Cleaning	Whitespace/case normalization, canonical mappings, datetime parsing, time zones
Large Datasets	Chunked reading, sampling for design, vectorization over loops, performance measurement
Full Pipeline	Modular design, order of operations (types→missing→outliers→strings→dates), logging, validation
Mini Project	End-to-end cleaning with documentation, verification, and analysis-ready output

Key Takeaway

Production data cleaning requires explicit strategies, documented rationale, and validation. Every decision affects model performance, fairness, and reproducibility.

```
clean_data(df) → clean_types() → clean_missing() → handle_outliers() → clean_strings_and_dates() →
validate() → ✓
```