Lecture 18 Handout

Data Cleaning Fundamentals with pandas

Transforming Messy Data into Analysis-Ready Datasets

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 7, Section 7.14 (continued) - Data Cleaning with pandas
Reference Notebooks: `ch07/07_14.ipynb` for data cleaning examples

Learning Objectives

By the end of this lecture, you will be able to:

1. **Detect Missing Data** using isna(), notna(), and info() to identify missing values in DataFrames
2. **Handle Missing Data** by applying dropna() and fillna() strategies appropriately for different scenarios
3. **Identify Duplicates** using drop_duplicates() to find and remove duplicate rows
4. **Validate Data Types** by checking and converting data types using dtype, astype(), and to_numeric()
5. **Standardize Data** by renaming columns for clarity and consistency using rename()
6. **Build Data Cleaning Workflows** by combining multiple cleaning operations into systematic pipelines
7. **Make Informed Decisions** by choosing appropriate cleaning strategies based on data context and analysis goals

Prerequisites Review

Building on Your pandas Foundation:

From Lectures 1-11, you have complete Python fundamentals including control flow, functions, lists, dictionaries, and file I/O. From Lectures 12-13, you mastered NumPy arrays with all array operations, indexing, boolean selection, and statistical functions.

From Lecture 16, you mastered pandas Series: creation with custom indices, accessing elements with .loc[] and .iloc[], statistical methods (count(), mean(), std(), describe()), string operations with .str accessor, and understanding NaN for missing data.

From Lecture 17, you mastered pandas DataFrames: understanding DataFrames as collections of aligned Series columns, creating DataFrames from dictionaries and lists, accessing data with .loc[] and .iloc[] for rows and columns, boolean indexing for filtering rows, adding and removing columns, statistical methods with axis parameter, and sorting with sort_values().

You've seen NaN in Series arithmetic when indices don't match, and you've used boolean masks extensively for filtering data. These concepts become the foundation for systematic data cleaning.

Transformation Goal: Evolve from **working with clean, perfect DataFrames** to **mastering real-world messy data with missing values, duplicates, inconsistent types, and quality issues** through systematic cleaning workflows.

1 Part 1: Introduction to Data Cleaning

1.1 Why Real-World Data is Always Messy

Think about organizing a messy room. Before you can find anything or use the space effectively, you need to clean up - remove trash, organize items, fix broken things, and create a system. Data cleaning is exactly the same process. Real-world data arrives messy, incomplete, inconsistent, and error-prone. Before you can analyze it, you need to clean it systematically.

Consider a student enrollment database. Registration systems allow duplicate entries when students accidentally submit forms twice. Not all fields are required, so some students have missing phone numbers or email addresses. Data entry clerks might type grades as ”A”, ”B”, ”C” instead of numeric values, breaking calculations. One clerk capitalizes column names (”Student_Name”), another uses lowercase (”student_name”), creating confusion. A sensor recording daily temperatures might fail occasionally, leaving gaps in your dataset. Survey responses often have missing answers for optional questions.

These aren't unusual scenarios - they're the norm. Every dataset you encounter in practice has quality issues. The phrase ”garbage in, garbage out” captures the fundamental truth: no matter how sophisticated your analysis techniques are, if your input data is flawed, your results will be meaningless or misleading. Data cleaning isn't a preliminary chore you rush through - it's often the most time-consuming and critical phase of any data analysis project, typically consuming 60-80% of a data scientist's time.

1.2 The Data Cleaning Workflow

Professional data cleaning follows a systematic four-phase workflow, not random fixes. First, you **Detect** problems by examining your data for missing values, duplicates, inconsistent types, and anomalies. Second, you **Assess** the scope and impact of these problems - how many rows affected? How severe is the issue? Third, you **Clean** the data by applying appropriate strategies - sometimes you remove problematic data, sometimes you fill missing values, sometimes you convert types or standardize formats. Finally, you **Verify** that your cleaning worked correctly and didn't introduce new problems.

This lecture focuses on the fundamental cleaning operations every data analyst must master: detecting and handling missing data, identifying and removing duplicates, validating and converting data types, and standardizing column names. By the end, you'll build complete cleaning workflows that transform messy real-world data into analysis-ready datasets.

2 Part 2: Detecting Missing Data

2.1 Understanding Missing Data

Missing data represents absence of values in your dataset. In pandas, missing data appears as NaN (Not a Number), a special value that means ”no data here.” Missing data occurs for many legitimate reasons: sensors fail temporarily, survey respondents skip optional questions, data merging creates gaps when records don't match, manual data entry has incomplete fields, or values simply aren't available yet.

The crucial insight is that missing data is information, not an error. When you see NaN in a temperature dataset, it tells you the sensor failed that day. When you see NaN in a survey response, it tells you the person chose not to answer. This information about data availability is valuable for understanding your data's completeness and reliability. pandas handles NaN values specially - most operations automatically ignore them rather than causing errors, which is usually the behavior you want.

2.2 Detecting Missing Values with isna() and notna()

The first step in handling missing data is detecting where it occurs. pandas provides `isna()` to create a boolean mask showing True for missing values and False for present values. Its complement `notna()` does the opposite - True for present values, False for missing ones.

```python
import pandas as pd
import numpy as np

# Student grades with some missing test scores
grades = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve'],
    'Test1': [85, 92, np.nan, 95, 88],
    'Test2': [87, np.nan, 90, 88, 91],
    'Test3': [np.nan, 89, 88, np.nan, 92]
})

# Boolean mask showing where values are missing
print("Missing data mask:")
print(grades.isna())

# Count missing values per column
print("\nMissing values per column:")
print(grades.isna().sum())
```

The boolean mask shows exactly which cells contain NaN. Summing this mask counts missing values because True equals 1 and False equals 0 in arithmetic operations. You can immediately see that Test1 has 1 missing value, Test2 has 1 missing value, and Test3 has 2 missing values. This quantifies the problem scope, helping you decide how to handle it.

2.3 Using info() for Data Overview

For a quick overview of your DataFrame's structure and missing data, the `info()` method is invaluable. It shows data types, non-null counts, and memory usage in one compact summary, making it perfect for initial data assessment.

```python
# Get comprehensive DataFrame information
print("DataFrame Information:")
print(grades.info())

# Calculate percentage of missing data per column
total_rows = len(grades)
missing_pct = (grades.isna().sum() / total_rows * 100)
```

```python
print("\nPercentage missing per column:")
print(missing_pct)
```

The `info()` output tells you ”5 non-null” for columns with complete data and fewer than 5 for columns with missing values. By comparing non-null counts to total rows, you instantly identify problematic columns. Calculating percentages helps assess severity - 5% missing might be acceptable, but 50% missing suggests the column might not be usable. This is your first checkpoint: before proceeding with analysis, always check what percentage of your data is actually present.

3 Part 3: Handling Missing Data - dropna()

3.1 When to Remove Missing Data

Sometimes the best strategy for handling missing data is removing it entirely. This makes sense when missing data is minimal (typically less than 5% of your dataset), when missing values make analysis impossible (you can't calculate a grade with no test scores), or when you have plenty of remaining data after removal so losing some rows doesn't hurt your analysis.

The trade-off with removal is obvious - you lose data and potentially lose information. If you have 100 students and 3 are missing ALL test scores, removing those 3 rows preserves 97% of your data with no complications. But if 50 students are each missing ONE test score, removing all 50 rows loses half your data unnecessarily. Understanding this trade-off is crucial for making good cleaning decisions.

3.2 Basic dropna() - Removing Any Missing Values

The simplest approach removes any row containing at least one missing value. This is aggressive but guarantees completely clean data with no gaps.

```python
# Student grades with scattered missing values
grades = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve'],
    'Test1': [85, 92, np.nan, 95, 88],
    'Test2': [87, np.nan, 90, 88, 91]
})

print("Original data:")
print(grades)
print(f"Shape: {grades.shape}")

# Remove rows with any missing values
clean_grades = grades.dropna()
print("\nAfter dropna():")
print(clean_grades)
print(f"Shape: {clean_grades.shape}")
```

This removes Charlie (missing Test1) and Bob (missing Test2), leaving only three students. Notice we went from 5 rows to 3 rows - we lost 40% of our data. For this dataset, `dropna()` might be too aggressive. But if those missing values truly make analysis impossible, it's the right choice.

3.3 dropna(how='all') - Removing Completely Empty Rows

A more conservative approach removes only rows where ALL values are missing. This preserves rows with partial data while eliminating truly empty records.

```python
# Dataset with some completely empty rows
data = pd.DataFrame({
    'A': [1, np.nan, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [9, np.nan, np.nan, 12]
})

print("Original with empty rows:")
print(data)

# Remove only completely empty rows
cleaned = data.dropna(how='all')
print("\nAfter dropna(how='all'):")
print(cleaned)
```

Row 1 and row 2 are completely empty (all NaN) and get removed. Row 0 and row 3 have data and are preserved. This is much less destructive than default `dropna()` - you only lose rows that contribute zero information.

3.4 dropna(subset=[]) - Selective Column Requirements

Often, some columns are critical (must have data) while others are optional. The `subset` parameter lets you specify which columns must have data for a row to be kept.

```python
# Student grades where Math and English are required
# but Science is optional
grades = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Math': [87, 92, np.nan, 95],
    'English': [90, 85, 92, np.nan],
    'Science': [np.nan, 88, 90, 92]
})

print("Original grades:")
print(grades)

# Keep students who have both Math AND English scores
# Allow missing Science scores
required = grades.dropna(subset=['Math', 'English'])
print("\nStudents with Math and English:")
print(required)
```

Charlie is removed (missing Math), Diana is removed (missing English), but Alice and Bob are kept even though Alice is missing Science. This flexibility lets you enforce business rules - perhaps Science is an elective course, so missing Science scores are acceptable, but Math and English are required courses.

4 Part 4: Handling Missing Data - fillna()

4.1 When to Fill Rather Than Drop

Filling missing values (imputation) preserves your data size but introduces assumptions about what the missing values would have been. Choose filling over dropping when missing data represents a significant portion of your dataset (more than 5-10%), when you can make reasonable assumptions about missing values based on context, or when maintaining data size is critical for your analysis.

The key is choosing an appropriate fill strategy based on your data's characteristics and meaning. Numbers and categories require different strategies. Normally distributed data suggests different fills than skewed data. Sequential data (like time series) has different options than independent observations. Let's explore the main filling strategies and when to use each.

4.2 Filling with a Constant Value

The simplest fill strategy uses a constant value for all missing data. This makes sense when the constant has clear meaning in your context - zero for missing quantities, "Unknown" for missing categories, or a default value specified by business rules.

```python
# Quiz scores where missing means student didn't submit
quiz_scores = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Quiz1': [85, 92, np.nan, 95],
    'Quiz2': [87, np.nan, 90, 88]
})

print("Original quiz scores:")
print(quiz_scores)

# Fill missing quizzes with 0 (not submitted = zero)
filled = quiz_scores.fillna(0)
print("\nAfter filling with 0:")
print(filled)

# Calculate averages
filled['Average'] = filled[['Quiz1', 'Quiz2']].mean(axis=1)
print("\nWith averages:")
print(filled)
```

This treats missing quizzes as zeros, which is appropriate if your grading policy says "not submitted equals zero credit." Charlie's Quiz1 and Bob's Quiz2 become 0, affecting their averages. This is honest and follows the policy, but be careful - filling with 0 is only appropriate when 0 genuinely means something in your context. For sensor readings or scientific measurements, 0 might mean something completely different than "missing."

4.3 Filling with Statistical Measures

For numeric data, filling with the mean (average) or median (middle value) is often appropriate. Mean filling works well for normally distributed data with few outliers. Median filling is better for skewed data or data with outliers because the median is robust to extreme values.

```python
# Temperature data with some missing readings
temps = pd.DataFrame({
    'Day': ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
    'Temperature': [72, np.nan, 75, np.nan, 69]
})

print("Original temperatures:")
print(temps)

# Calculate mean before filling
temp_mean = temps['Temperature'].mean()
print(f"\nMean temperature: {temp_mean:.1f}")

# Fill missing values with mean
temps['Filled_Mean'] = temps['Temperature'].fillna(temp_mean)
print("\nAfter filling with mean:")
print(temps)

# Alternative: fill with median
temp_median = temps['Temperature'].median()
```

```python
temps['Filled_Median'] = temps['Temperature'].fillna(temp_median)
print(f"\nMedian temperature: {temp_median:.1f}")
print(temps)
```

Tuesday and Thursday temperatures were missing. Filling with the mean (72.0) assumes missing days had average conditions. For temperature data, this is often reasonable - it's unlikely the sensor failed on particularly unusual days. However, note that mean filling can't introduce outliers (you're filling with the average by definition), while median filling is even more conservative. Choose mean when you want to preserve the overall statistical properties, median when you're concerned about outliers affecting the mean.

4.4 Forward Fill and Backward Fill

For sequential data where adjacent values are related (time series, sensor readings, sequential measurements), forward fill uses the previous value and backward fill uses the next value. This assumes gradual change rather than sudden jumps.

```python
# Daily temperatures with some missing readings
daily_temps = pd.DataFrame({
    'Day': ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
    'Temperature': [72, np.nan, 75, np.nan, 69]
})

print("Original temperatures:")
print(daily_temps)

# Forward fill - use previous day's temperature
forward = daily_temps['Temperature'].fillna(method='ffill')
print("\nForward fill:")
print(forward)

# Backward fill - use next day's temperature
backward = daily_temps['Temperature'].fillna(method='bfill')
print("\nBackward fill:")
print(backward)
```

Forward fill gives Tuesday 72 (Monday's temperature) and Thursday 75 (Wednesday's temperature). This assumes temperature persists until you measure a change. Backward fill gives Tuesday 75 (Wednesday's temperature) and Thursday 69 (Friday's temperature). This assumes the next measurement reveals what the missing day was like. Forward fill is more common because we typically know the past but not the future. However, these methods only make sense for sequential data where neighboring values are related - don't use them for independent observations like survey responses.

4.5 Column-Specific Fill Strategies

Different columns often need different fill strategies based on what they represent. Numeric columns might use mean or median, categorical columns might use mode or ”Unknown”, counts and quantities might use 0, and sequential data might use forward fill. Professional data cleaning applies appropriate strategies per column.

```python
# Sales data with mixed types of missing values
sales = pd.DataFrame({
    'Product': ['Widget', 'Gadget', 'Tool', 'Item'],
    'Price': [19.99, np.nan, 29.99, 24.99],
    'Quantity': [100, 50, np.nan, 75],
    'Category': ['A', 'B', np.nan, 'A']
})

```

```python
print("Original sales data:")
print(sales)

# Fill Price with median (robust to outliers)
sales['Price'] = sales['Price'].fillna(sales['Price'].median())

# Fill Quantity with 0 (missing quantity = none sold)
sales['Quantity'] = sales['Quantity'].fillna(0)

# Fill Category with "Unknown" (missing category)
sales['Category'] = sales['Category'].fillna('Unknown')

print("\nAfter tailored filling:")
print(sales)
```

Each column gets a strategy appropriate for its type and meaning. Price uses median because prices might be skewed (a few very expensive items). Quantity uses 0 because it's a count and 0 means "nothing." Category uses "Unknown" because that's honest labeling for missing categorical data. This demonstrates the principle: understand your data's meaning and context, then choose fill strategies accordingly.

5 Part 5: Identifying and Removing Duplicates

5.1 What are Duplicate Rows?

Duplicate rows are multiple records that represent the same entity or event appearing more than once in your dataset. They occur due to data entry errors (accidentally submitting a form twice), system glitches (database transaction failures), or data integration issues (merging datasets creates overlapping records). Duplicates distort analysis by over-counting entities, creating bias in statistics, and giving extra weight to repeated records.

The crucial skill is distinguishing true duplicates from legitimate repeated values. If you have two students both named "John Smith" with different student IDs, they're NOT duplicates - they're two different people who happen to share a name. But if you have the same student ID appearing twice with slightly different name spellings (Alice Smith vs Alice M. Smith), those ARE duplicates - same person entered twice. Duplicate detection depends on identifying unique identifier columns.

5.2 Detecting Duplicates with duplicated()

The `duplicated()` method identifies duplicate rows by returning a boolean Series marking which rows are duplicates of earlier rows. By default, the first occurrence is marked False and subsequent occurrences are marked True.

```python
# Student enrollment with duplicate entries
students = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob'],
    'ID': [101, 102, 101, 103, 104],
    'Course': ['Math', 'English', 'Math', 'Science', 'History']
})

print("Student enrollments:")
print(students)

# Find duplicate rows (all columns identical)
dupes = students.duplicated()
print("\nDuplicate mask:")
print(dupes)

```

```python
# Show which rows are duplicates
print("\nDuplicate rows:")
print(students[dupes])

# Count total duplicates
print(f"\nTotal duplicates: {dupes.sum()}")
```

Row 2 is marked as duplicate because it's identical to row 0 (Alice, 101, Math). The first occurrence (row 0) is not marked as duplicate, only the second occurrence (row 2) is. Row 4 is NOT marked as duplicate even though the name "Bob" repeats, because the ID is different (104 vs 102) - these are different students, not duplicates.

5.3 Removing Duplicates with drop_duplicates()

Once you've identified duplicates, `drop_duplicates()` removes them. The `keep` parameter controls which occurrence to keep: 'first' keeps the first occurrence (default), 'last' keeps the last occurrence, or False removes all occurrences of duplicated rows.

```python
# Student data with duplicates
students = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'Charlie'],
    'ID': [101, 102, 101, 103],
    'GPA': [3.8, 3.6, 3.9, 3.7]
})

print("Original with duplicates:")
print(students)

# Keep first occurrence of duplicates
first = students.drop_duplicates(keep='first')
print("\nKeep first:")
print(first)

# Keep last occurrence of duplicates
last = students.drop_duplicates(keep='last')
print("\nKeep last:")
print(last)
```

Alice appears twice (row 0 and row 2). When keeping first, we keep row 0 (GPA 3.8). When keeping last, we keep row 2 (GPA 3.9). Which is correct? It depends on your data. If the later entry is more recent and accurate, keep last. If the first entry is the original authoritative record, keep first. Often you need to examine the actual data to decide which duplicate to keep.

5.4 Subset-Based Duplicate Detection

Usually, you don't want to check if ALL columns are identical - you want to check if key identifier columns are identical. A student appearing twice with the same ID but different GPAs (one old, one updated) should be considered a duplicate based on ID alone. The `subset` parameter specifies which columns define uniqueness.

```python
# Student records where ID is the unique identifier
students = pd.DataFrame({
    'ID': [101, 102, 101, 103, 102],
    'Name': ['Alice', 'Bob', 'Alice M.', 'Charlie', 'Robert'],
    'GPA': [3.8, 3.6, 3.9, 3.7, 3.6]
})

print("Student records:")
print(students)

```

```python
# Find duplicates based on ID only
id_dupes = students.duplicated(subset=['ID'])
print("\nDuplicates by ID:")
print(students[id_dupes])

# Remove duplicates based on ID, keep last (most recent)
unique = students.drop_duplicates(subset=['ID'], keep='last')
print("\nUnique students (by ID):")
print(unique)
```

Row 2 is a duplicate of row 0 (both ID 101) and row 4 is a duplicate of row 1 (both ID 102), even though names differ slightly. This is correct - the same student ID with different name spellings represents one person with data entry variations. By keeping 'last', we assume the most recent entry has the correct information. This is the professional approach: identify your unique key columns (ID, email, transaction number) and detect duplicates based on those keys.

6 Part 6: Data Type Validation and Conversion

6.1 Why Data Types Matter

Data types determine what operations are possible on your data. When pandas reads a CSV file, it tries to infer types automatically, but it sometimes guesses wrong. Numbers stored as strings can't be summed or averaged. Dates stored as strings can't be sorted chronologically. Categories stored as numbers waste memory and confuse analysis. Type validation catches these issues before they cause mysterious errors or incorrect results.

Think of data types as the vocabulary your analysis speaks. If you try to ask pandas "what's the average price?" but prices are stored as strings like "19.99", *pandascan′tanswer − itdoesn′tknowhowtoaveragetext.Convertingpricestonumerictypesletspandasperformcalculationscorrectly.T convertingdataintothelanguageyouranalysistoolsunderstand.*

6.2 Checking Current Data Types

Before converting types, check what types you currently have. The `dtypes` attribute shows each column's type, and the `info()` method shows types alongside other useful information.

```python
# Sales data with mixed types
sales = pd.DataFrame({
    'Product': ['Widget', 'Gadget', 'Tool'],
    'Price': ['19.99', '29.99', '24.99'],  # Strings!
    'Quantity': [100, 50, 75],
    'InStock': ['True', 'False', 'True']  # Strings!
})

print("Sales data:")
print(sales)

# Check data types
print("\nData types:")
print(sales.dtypes)

# Try to calculate total price (will fail!)
try:
    total = sales['Price'].sum()
    print(f"\nTotal: {total}")
except TypeError as e:
    print(f"\nError: {e}")
```

The output shows Price is 'object' (pandas' name for string), not numeric. When you try to sum prices, you get an error or unexpected results (string concatenation instead of addition). This demonstrates why type validation matters - operations fail silently or produce wrong answers when types are incorrect.

6.3 Converting Types with astype()

For clean data where you're confident conversion will succeed, `astype()` directly converts to a specified type. This is fast and straightforward when you know the data is valid.

```python
# Sales data that needs type conversion
sales = pd.DataFrame({
    'Product': ['Widget', 'Gadget', 'Tool'],
    'Price': ['19.99', '29.99', '24.99'],
    'Quantity': ['100', '50', '75'],
    'InStock': ['True', 'False', 'True']
})

print("Original types:")
print(sales.dtypes)

# Convert Price from string to float
sales['Price'] = sales['Price'].astype(float)

# Convert Quantity from string to int
sales['Quantity'] = sales['Quantity'].astype(int)

# Convert InStock from string to bool
sales['InStock'] = sales['InStock'].astype(bool)

print("\nAfter conversion:")
print(sales.dtypes)

# Now calculations work correctly
print(f"\nAverage price: ${sales['Price'].mean():.2f}")
print(f"Total quantity: {sales['Quantity'].sum()}")
```

After conversion, Price is float64, Quantity is int64, and InStock is bool. Now mathematical operations work correctly - you can calculate averages, sums, and perform logical operations. This is the clean path when you know your data is well-formatted.

6.4 Safe Conversion with to_numeric()

Real-world data is rarely clean enough for `astype()`. Some values might not convert correctly - perhaps "Call for pricing" appears in a price column, or "N/A" appears in a numeric field. The `to_numeric()` function with `errors='coerce'` handles conversion failures gracefully by converting invalid values to NaN.

```python
# Messy price data with non-numeric values
prices = pd.DataFrame({
    'Item': ['Widget', 'Gadget', 'Tool', 'Item'],
    'Price': ['19.99', 'Call', '29.99', '24.99']
})

print("Original prices:")
print(prices)

# Safe conversion - failures become NaN
prices['Price'] = pd.to_numeric(prices['Price'],
                                errors='coerce')

```

```python
print("\nAfter safe conversion:")
print(prices)

# Identify which values failed conversion
failed = prices['Price'].isna()
print(f"\nFailed conversions:")
print(prices[failed])

# Calculate mean, ignoring failed conversions
print(f"\nAverage (ignoring NaN): ${prices['Price'].mean():.2f}")
```

The ”Call” entry can’t convert to a number, so it becomes NaN. This is much better than raising an error and stopping your entire workflow. Now you know exactly which values had problems (those with NaN after conversion), and you can decide whether to fill them, drop them, or investigate further. The mean calculation automatically ignores NaN, giving you the average of valid prices.

7 Part 7: Renaming Columns and Complete Workflow

7.1 Standardizing Column Names

Column names are your data’s labels - they should be clear, consistent, and easy to work with in code. Real datasets often have messy column names with spaces, inconsistent capitalization, abbreviations, or special characters. Standardizing column names makes your code readable and prevents errors.

```python
# Dataset with messy column names
data = pd.DataFrame({
    'Student Name': ['Alice', 'Bob', 'Charlie'],
    'test_score': [87, 92, 88],
    'FinalGrade': [90, 88, 91]
})

print("Original column names:")
print(data.columns)

# Rename columns to consistent format
data = data.rename(columns={
    'Student Name': 'student_name',
    'test_score': 'test_score',  # Already good
    'FinalGrade': 'final_grade'
})

print("\nStandardized column names:")
print(data.columns)
print("\n", data)
```

After renaming, all columns follow snake_case convention (lowercase with underscores) with no spaces. This consistency makes your code cleaner and prevents bugs that arise from typos in column names. Professional datasets always have standardized, descriptive column names.

7.2 Complete Data Cleaning Workflow

Let’s combine everything we’ve learned into a complete data cleaning workflow that demonstrates the systematic approach professionals use. This example starts with messy data and applies all cleaning techniques in a logical order.

```python
import pandas as pd
import numpy as np

```

```python
# Messy student dataset (realistic problems)
messy_data = pd.DataFrame({
    'Student Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Diana'],
    'ID': [101, 102, 101, 103, np.nan],
    'Math': ['87', np.nan, '87', '78', '95'],
    'English': ['90', '85', np.nan, '82', '88'],
    'Science': [85, 88, 85, np.nan, 92]
})

print("STEP 1: Inspect original data")
print(messy_data)
print(f"\nShape: {messy_data.shape}")
print("\nData types:")
print(messy_data.dtypes)

print("\n" + "="*50)
print("STEP 2: Detect data quality issues")
print(f"Missing values per column:")
print(messy_data.isna().sum())
print(f"\nDuplicates found: {messy_data.duplicated().sum()}")

print("\n" + "="*50)
print("STEP 3: Remove duplicates (keep first occurrence)")
clean_data = messy_data.drop_duplicates()
print(f"Shape after removing duplicates: {clean_data.shape}")

print("\n" + "="*50)
print("STEP 4: Convert data types")
clean_data['Math'] = pd.to_numeric(clean_data['Math'],
                                   errors='coerce')
clean_data['English'] = pd.to_numeric(clean_data['English'],
                                      errors='coerce')
print("Data types after conversion:")
print(clean_data.dtypes)

print("\n" + "="*50)
print("STEP 5: Handle missing data")
# Drop row if ID is missing (critical field)
clean_data = clean_data.dropna(subset=['ID'])
# Fill missing grades with column median
clean_data['Math'] = clean_data['Math'].fillna(
    clean_data['Math'].median())
clean_data['English'] = clean_data['English'].fillna(
    clean_data['English'].median())
clean_data['Science'] = clean_data['Science'].fillna(
    clean_data['Science'].median())

print("\n" + "="*50)
print("STEP 6: Rename columns for consistency")
clean_data = clean_data.rename(columns={
    'Student Name': 'student_name'
})

print("\n" + "="*50)
print("FINAL CLEANED DATA:")
print(clean_data)
print(f"\nFinal shape: {clean_data.shape}")
print("No missing values:", clean_data.isna().sum().sum() == 0)
```

This workflow demonstrates the professional approach: inspect the raw data first to understand its problems, detect and quantify issues (missing data, duplicates), remove duplicates to avoid over-counting, convert types so operations work correctly, handle missing data with appropriate strategies for each column, standardize column names for consistency, and verify the cleaned result. Each step builds on the previous one, and you check your work at each stage. This is the template you'll use for every data cleaning task.

Summary and Best Practices

Key Takeaways:

1. Real-world data is always messy - missing values, duplicates, wrong types, and inconsistencies are normal
2. Detect missing data with isna(), notna(), and info() before deciding how to handle it
3. Choose dropna() for minimal missing data (<5%) or critical fields; choose fillna() to preserve data size
4. Fill strategies depend on data type and context: mean/median for numeric, forward fill for sequential, constants for categories
5. Identify duplicates with duplicated(subset=[]) using unique identifier columns, not all columns
6. Validate data types with dtypes and convert safely using to_numeric(errors='coerce')
7. Follow systematic workflow: Detect → Assess → Clean → Verify at every step
8. Document your cleaning decisions - they're part of your analysis methodology

Common Pitfalls to Avoid

- Using dropna() without checking how much data you'll lose
- Filling with mean without checking if data is skewed (use median for skewed data)
- Using forward/backward fill on non-sequential data where neighboring values aren't related
- Detecting duplicates based on all columns instead of unique identifier columns
- Using astype() on messy data that might have conversion failures (use to_numeric with errors='coerce')
- Forgetting to verify cleaned data before proceeding with analysis
- Not documenting which cleaning strategies you applied and why

Practice Exercises

Exercise 1: Survey Data Cleaning
Given survey data with many missing responses (people skipped optional questions), detect which questions have the most missing data, decide appropriate handling for required vs optional fields, fill numeric ratings with median, fill text responses with "No response", and remove surveys with all fields missing.

Exercise 2: Sales Data Quality
Given sales data with duplicate transactions, some negative quantities (data errors), prices stored as strings with $ symbols, and missing product categories, identify and remove duplicate

transaction IDs keeping the most recent, fix negative quantities (replace with positive values), convert prices to numeric type handling $ symbols, and fill missing categories with ”Uncategorized”.

Exercise 3: Sensor Data Cleaning
Given temperature sensor data with equipment failures causing missing readings and some impossible values (500 degrees), detect missing data patterns, use forward fill for short gaps (1-2 readings) but drop longer gaps, identify outliers using statistical methods (values beyond 3 standard deviations), and replace outliers with median temperature.

Next Lecture Preview

In Lecture 19, we’ll advance to more sophisticated data manipulation techniques including reading data from external files (CSV, Excel), combining DataFrames with merge and join operations, grouping and aggregation with groupby(), and reshaping data with pivot tables. The data cleaning skills you’ve mastered today become the foundation for these advanced operations - you’ll always clean your data first before performing complex transformations and analysis.