Pandas — Complete Data Cleaning & Visualization Reference

Comprehensive, print-friendly guide: everything essential for cleaning data with Pandas (with examples & syntax)

Overview — Data Cleaning with Pandas

Data cleaning is the process of detecting, correcting, and preparing data for analysis. This guide collects the core Pandas functions, idioms, and code snippets used in real-world data cleaning workflows — from detecting missing values to handling dates, outliers, strings, categorical encoding, and visual checks.

```
# Recommended imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv("data.csv")
df.head()
```

Tip: keep a reproducible notebook or script with the data-cleaning steps (so you can re-run preprocessing whenever datasets update).

Missing Data — Detection & Handling

Missing values are one of the most common problems. Pandas provides many tools for detecting and handling them.

Detect missing values

```
df.isna()           # boolean DataFrame of missing values
df.isnull().sum()   # count per column
df.isna().mean()    # proportion of missing per column
df.info()           # quick summary showing non-null counts
```

Use `df.isna()` and aggregations to determine which columns need attention.

Drop missing values

```
# Drop rows with any missing values
df_clean = df.dropna(axis=0, how='any')

# Drop rows that have missing values in specific columns
df_clean = df.dropna(subset=['col1','col2'])

# Drop columns with too many missing values
df.dropna(axis=1, thresh=int(0.6*len(df)), inplace=True)  # keep cols with >=60% non-null
```

Parameter	Meaning
axis	0 drop rows, 1 drop columns
how	'any' (drop if any NA), 'all' (drop if all NA)
subset	list of columns to check
thresh	require at least thresh non-NA values

Fill missing values

Choose filling strategy depending on column type and context.

```
# Fill with scalar
df['Age'] = df['Age'].fillna(0)

# Fill numeric with column mean/median
df['Salary'] = df['Salary'].fillna(df['Salary'].median())

# Forward/backward fill (time-series or ordered data)
df.sort_values('Date', inplace=True)
```

```
df['Value'] = df['Value'].fillna(method='ffill')  # forward fill

# Fill per-column with a dict
df.fillna({'Age': df['Age'].median(), 'City': 'Unknown'}, inplace=True)
```

Parameter	Meaning
value	scalar or dict per column
method	'ffill' or 'bfill' (forward/backward fill)
limit	max consecutive fills

Replace & interpolate

```
# Replace specific sentinel values with NaN, then handle them
df.replace({'': np.nan, 'NA': np.nan, -1: np.nan}, inplace=True)

# Interpolate numeric columns (linear/time)
df['temperature'] = df['temperature'].interpolate(method='linear', limit_direction='both')
```

Use interpolation for numeric series where values are missing in short sequences and a smooth estimate is reasonable.

Data Types & Conversion

Correct types are essential (numbers as numeric, dates as datetime, categories as category dtype). Converting early prevents subtle bugs.

Inspect types

```
df.dtypes
df.select_dtypes(include='object').columns  # textual columns
df.select_dtypes(include=['number']).columns
```

Convert types

```
# Convert to numeric (coerce invalid -> NaN)
df['col'] = pd.to_numeric(df['col'], errors='coerce')

# Convert to datetime
df['date'] = pd.to_datetime(df['date'], errors='coerce', dayfirst=False, infer_datetime_format=Tru

# Convert to timedelta
df['duration'] = pd.to_timedelta(df['duration'], errors='coerce')

# Convert to category for memory & semantics
df['color'] = df['color'].astype('category')

# Generic astype
df = df.astype({'id':'int64','price':'float64'})
```

Use `errors='coerce'` to turn invalid parsing into NaN and then handle them explicitly.

Safe conversion helpers

```
# Attempt conversion with fallback
def to_int_safe(x):
    try:
        return int(x)
    except:
        return np.nan

df['col_int'] = df['col'].apply(to_int_safe)
```

Duplicates — Detecting & Removing

Duplicates can arise from bad merges, repeated scraping, or logging issues.

Detect duplicates

```
# Boolean mask of duplicates (first occurrence kept)
mask = df.duplicated()
df[mask]

# Duplicates based on specific subset of columns
df[df.duplicated(subset=['name','email'])]
```

Drop duplicates

```
# Remove duplicates, keep first (default)
df = df.drop_duplicates()

# Keep last occurrence
df = df.drop_duplicates(subset=['name','email'], keep='last')

# Drop all rows that have duplicates (keep=False)
df = df.drop_duplicates(subset=['name','email'], keep=False)
```

Outliers & Scaling

Outlier handling depends on whether outliers are errors or informative. Always visualize before removing.

IQR method

```
Q1 = df['col'].quantile(0.25)
Q3 = df['col'].quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR

outliers = df[(df['col'] < lower) | (df['col'] > upper)]
non_outliers = df[(df['col'] >= lower) & (df['col'] <= upper)]
```

Z-score method

```
from scipy import stats
z_scores = np.abs(stats.zscore(df['col'].dropna()))
mask = (z_scores > 3)  # typical threshold
outliers = df.loc[df['col'].dropna().index[mask]]
```

Clipping & Winsorizing

```
# Clip to bounds
df['col_clipped'] = df['col'].clip(lower=lower, upper=upper)

# Winsorize (using scipy) - cap extreme values rather than drop
from scipy.stats.mstats import winsorize
df['col_win'] = winsorize(df['col'], limits=[0.01, 0.01])  # 1% on each side
```

Scaling (for ML pipelines)

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
scaler = StandardScaler()
df[['num1','num2']] = scaler.fit_transform(df[['num1','num2']])
```

Do scaling after splitting train/test to avoid data leakage when building models.

String Cleaning & Text Normalization

Text columns often require trimming, casing, pattern extraction, and handling of inconsistent separators.

Basic string ops (.str)

```
# Lowercase/strip
df['name'] = df['name'].str.strip().str.lower()

# Replace substrings or regex
df['phone'] = df['phone'].str.replace(r'\D', '', regex=True)  # remove non-digits

# Check contains
mask = df['email'].str.contains('@', na=False)

# Split and extract
df['domain'] = df['email'].str.split('@').str[1]

# Extract with regex
df['zip'] = df['address'].str.extract(r'(\d{5})')
```

Normalize whitespace & punctuation

```
# Normalize multiple spaces to single, and strip
df['text'] = df['text'].str.replace(r'\s+', ' ', regex=True).str.strip()

# Replace unusual quotes or characters
df['text'] = df['text'].str.replace('“', '"').str.replace('”','"')
```

Tokenization & more (lightweight)

```
# Simple tokenization
df['tokens'] = df['text'].str.split()

# Count words
df['word_count'] = df['tokens'].str.len()
```

For advanced NLP cleaning use libraries such as `spaCy` or `nltk`.

Columns & Rows — Rename, Drop, Reorder, Impute

Rename & reorder

```
# Rename columns
df = df.rename(columns={'old_name':'new_name'})

# Rename in-place
df.rename(columns=str.lower, inplace=True)

# Reorder columns
cols = ['id','name','email','age']
df = df[cols]
```

Drop columns & rows

```
df.drop(['unnecessary_col'], axis=1, inplace=True)
df.drop(index=[0,1,2], inplace=True)   # drop by index labels
```

Insert & create columns

```
# Create derived column
df['age_group'] = pd.cut(df['age'], bins=[0,18,35,60,120], labels=['child','young','adult','senio

# Insert at position
df.insert(2, 'uid', range(1, len(df)+1))
```

Reindexing & resetting index

```
# Reset index after filtering
df = df.reset_index(drop=True)

# Set index to a column
df = df.set_index('id')

# Reindex to include missing rows (useful for timeseries)
new_index = pd.date_range(start='2020-01-01', end='2020-12-31', freq='D')
df = df.reindex(new_index)
```

Handling Inconsistencies — Mapping & Conditional Fixes

Map values & standardize categories

```
# Map variants to canonical values
city_map = {'ny': 'New York', 'nyc': 'New York', 'n.y.': 'New York'}
df['city_clean'] = df['city'].str.lower().map(city_map).fillna(df['city'].str.title())

# Use replace for many-to-many replacements
df['status'] = df['status'].replace({'Y':'Yes','N':'No','unknown':np.nan})
```

Conditional changes — where & mask

```
# Where: retain values where condition is True else replace
df['score_clean'] = df['score'].where(df['score'] >= 0, other=np.nan)

# Mask: replace values where condition is True
df.loc[df['age'] < 0, 'age'] = np.nan
```

Apply / applymap for custom fixes

```
# Elementwise on a Series
df['col'] = df['col'].apply(lambda x: x.strip() if isinstance(x, str) else x)

# Applymap over entire DataFrame (use carefully — slow)
df = df.applymap(lambda x: x.strip() if isinstance(x, str) else x)
```

Prefer vectorized Pandas string/Numpy ops when possible — they are faster than Python loops.

Categorical Data — Encoding & Categories

Category dtype

```
df['city'] = df['city'].astype('category')
df['city'].cat.categories    # list categories
df['city'].cat.codes         # numeric codes
```

One-hot encoding

```
# Using pandas
df = pd.get_dummies(df, columns=['color'], prefix='color', drop_first=False)

# Or for specific column
dummies = pd.get_dummies(df['size'], prefix='size')
df = pd.concat([df, dummies], axis=1)
```

Label encoding / factorize

```
# Simple numeric codes
df['city_code'], uniques = pd.factorize(df['city'])
```

Choose encoding based on model needs — one-hot for nominal small-cardinality features, ordinal/coded for ordered categories.

Date & Time Cleaning

Convert to datetime

```
df['date'] = pd.to_datetime(df['date_str'], errors='coerce', infer_datetime_format=True)

# If day-first date strings
df['date'] = pd.to_datetime(df['date_str'], dayfirst=True, errors='coerce')
```

Use the .dt accessor

```
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['weekday'] = df['date'].dt.day_name()
df['is_weekend'] = df['date'].dt.weekday >= 5
```

Resampling & filling time gaps

```
# Set index to datetime
df_ts = df.set_index('date').sort_index()

# Resample to daily, filling forward
df_ts.resample('D').mean().ffill()

# Rolling window example
df_ts['7day_avg'] = df_ts['value'].rolling(window=7).mean()
```

Data Integration — Merge, Join & Concat

pd.merge (SQL-style joins)

```
# Inner join
pd.merge(df_left, df_right, how='inner', on='id')

# Left join with different key names
pd.merge(emp, dept, left_on='dept_id', right_on='id', how='left')

# Multiple keys
pd.merge(a, b, on=['id','date'], how='outer')
```

Concatenate & append

```
# Stack vertically
pd.concat([df1, df2], axis=0, ignore_index=True)

# Side-by-side
pd.concat([df1, df2], axis=1)
```

Join using index

```
# Join on index
df_left.join(df_right, how='left')
```

Be explicit about how and check for unintended cartesian products when keys are not unique.

Reshaping Data — Pivot, Melt, Stack, Unstack

Pivot / pivot_table

```
# Simple pivot
df.pivot(index='date', columns='region', values='sales')

# pivot_table with aggregation (handles duplicates)
df.pivot_table(index='date', columns='region', values='sales', aggfunc='sum', fill_value=0)
```

Melt (wide → long)

```
df_long = df.melt(id_vars=['id','date'], value_vars=['var1','var2'], var_name='variable', value_n
```

Stack / Unstack

```
stacked = df.set_index(['id','date']).stack()
unstacked = stacked.unstack(level=-1)
```

Visualization — Quick Visual Checks for Cleaning

Visual checks are essential: missingness maps, boxplots for outliers, histograms for distributions, scatter for relationships.

Missingness visualization

```
# Simple missingness heatmap using seaborn (if available)
import seaborn as sns
sns.heatmap(df.isna(), cbar=False)
plt.title('Missing values map')
plt.show()
```

Distribution checks

```
# Histogram
df['salary'].plot(kind='hist', bins=30)
plt.title('Salary distribution')
plt.xlabel('Salary')
plt.show()

# Boxplot for outliers
df.boxplot(column='salary')
plt.show()
```

Scatter & correlation

```
# Scatter plot
df.plot.scatter(x='age', y='salary')

# Correlation heatmap
sns.heatmap(df.select_dtypes(include='number').corr(), annot=True, fmt=".2f")
plt.show()
```

Time-series visual checks

```
# Line plot (after setting datetime index)
df_ts['value'].plot()
plt.title('Timeseries of value')
plt.show()
```

> Always plot before removing outliers or filling gaps — plots often reveal issues not obvious from tables.

Practical Data Cleaning Workflow Checklist

1. Inspect data: `df.head()`, `df.info()`, `df.describe()`, `df.shape`.
2. Identify and handle missing values (drop/fill/interpolate) based on context.
3. Correct data types (`to_numeric`, `to_datetime`, `astype('category')`).
4. Detect & resolve duplicates (`duplicated()`, `drop_duplicates()`).
5. Standardize text columns (.str methods): casing, whitespace, punctuation.
6. Fix inconsistent categories (map/replace), create canonical forms.
7. Handle outliers carefully (visualize, then decide: clip/winsorize/remove).
8. Encode categorical variables appropriately for downstream tasks.
9. Reshape/merge/concatenate as needed & validate joins (check row counts).
10. Save cleaned dataset (`df.to_csv('cleaned.csv', index=False)`) and document changes.

Example: Minimal reproducible cleaning script

```
import pandas as pd
import numpy as np

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

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

# 2. Replace sentinel values
df.replace({'': np.nan, 'n/a': np.nan, 'NA': np.nan, '-': np.nan}, inplace=True)

# 3. Convert types
df['date'] = pd.to_datetime(df['date'], errors='coerce')
df['amount'] = pd.to_numeric(df['amount'], errors='coerce')

# 4. Fill missing
df['amount'] = df['amount'].fillna(df['amount'].median())

# 5. Remove duplicates
df.drop_duplicates(subset=['transaction_id'], inplace=True)

# 6. Save cleaned file
df.to_csv("cleaned.csv", index=False)
```

Additional Tips & Resources

- Keep an immutable copy of the raw data; perform cleaning on a copied DataFrame.
- Use small exploratory scripts then encapsulate repeated logic into functions.
- Version your cleaned datasets (e.g., include date in filename) to track changes.
- For large datasets consider `dask.dataframe` or sampling for exploration.
- Consider writing tests for cleaning functions (unit tests for corner cases).

Suggested reading: Pandas official docs (user guide), "Python for Data Analysis" by Wes McKinney, and various data-cleaning notebooks on GitHub.