NumPy for Data Science — Complete Practical Guide

Concise, reproducible NumPy patterns for data cleaning, preparation, and numerical hygiene that every data scientist should know.

What is NumPy?

NumPy (Numeric Python) is the fundamental package for scientific computing in Python. It provides a fast, memory-efficient N-dimensional array object (`ndarray`), vectorized operations, broadcasting, and tools for numerical routines (linear algebra, random sampling, FFT). NumPy is the low-level engine under many data-science libraries (Pandas, SciPy, scikit-learn, TensorFlow).

NumPy focuses on homogeneous, typed arrays and high-performance computation — ideal for large numeric datasets and for prepping data before modeling or passing to higher-level libraries.

Why NumPy Matters in Data Science

NumPy matters because performance and memory efficiency are critical when working with large numeric datasets. Vectorized operations and broadcasting let you replace slow Python loops with fast C-backed implementations.

- **Speed:** vectorized ops run in compiled code — orders of magnitude faster than Python loops for large arrays.
- **Memory:** compact, typed arrays reduce memory overhead vs. Python lists.
- **Interoperability:** many ML libraries accept NumPy arrays directly (scikit-learn, Keras, XGBoost).
- **Deterministic numeric tools:** primitives for masks, indexing, and linear algebra are essential for preprocessing.

How NumPy Helps with Data Cleaning & Preparation

NumPy provides the building blocks for many cleaning tasks a data scientist performs on numeric arrays before feeding data into models or visualizations.

Primary categories

- **Missing value handling:** masks with `np.isnan`, sentinel replacement, and careful dtype choices.
- **Vectorized transforms:** apply arithmetic, scaling, normalization, and log transforms across arrays.
- **Filtering & boolean indexing:** select or remove rows based on numeric conditions quickly.
- **Outlier detection:** percentile-based or z-score methods with fast reductions.
- **Reshaping & alignment:** reshape, stack, split arrays; align multi-feature matrices for ML input.
- **Type conversion & numeric stability:** control precision (float32 vs float64), handle overflow/underflow, and manage infinities.

NumPy — Essential Techniques (with examples)

Below are the canonical NumPy patterns for cleaning, preparing, and transforming numeric data used in data science workflows.

1. Creating & inspecting arrays

```
import numpy as np
# Create arrays
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Inspect
arr.shape, arr.dtype, arr.ndim
np.unique(arr)  # unique values
```

Prefer NumPy arrays over Python lists when you need numeric computation at scale.

2. Missing values, masks & sentinels

```
# Use np.nan for missing floats
a = np.array([1.0, np.nan, 3.0, np.nan])
np.isnan(a)            # boolean mask of missing
np.nanmean(a)          # mean ignoring NaN

# Replace sentinel values (e.g., -999) with np.nan
a = np.array([1.0, -999.0, 3.0])
a[a == -999] = np.nan

# Remove rows with any NaN in a 2D array
M = np.array([[1.0, 2.0], [np.nan, 4.0], [5.0, 6.0]])
mask = ~np.isnan(M).any(axis=1)
clean = M[mask]
```

Note: NumPy arrays require a float dtype to store `np.nan`. For mixed types or labeled data, convert to Pandas.

3. Type conversion & coercion

```
# Convert type
x = np.array(["1", "2", "3"])
xi = x.astype(int)

# Safe numeric coercion with unknown values
from numpy import vectorize

def safe_int(s):
    try:
        return int(s)
    except:
        return np.nan

v_safe_int = np.vectorize(safe_int)
arr = v_safe_int(np.array(["1", "two", "3"]))
```

Use astype when you know values are clean; use vectorized safe parsers when inputs may be messy.

4. Vectorized operations & broadcasting

```
# Vectorized math (fast)
a = np.arange(1_000_000)
b = a * 2 + 3

# Broadcasting example: normalize per-column
X = np.random.rand(100, 3)
col_means = X.mean(axis=0)
X_centered = X - col_means  # broadcasting subtracts per-column
means
```

Always prefer vectorized code over Python loops for numeric arrays.

5. Boolean indexing & complex filters

```
# Simple filter
x = np.linspace(0, 10, 11)
```

```
mask = (x >= 2) & (x <= 8)
subset = x[mask]

# Multi-column filter on 2D array
M = np.random.randn(100, 4)
mask = (M[:,0] > 0) & (M[:,1] < 1.5)
rows = M[mask]
```

6. Outlier detection & handling

```
# IQR clipping per column
Q1 = np.nanpercentile(M, 25, axis=0)
Q3 = np.nanpercentile(M, 75, axis=0)
IQR = Q3 - Q1
lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR

# Clip values (keeps shape)
M_clipped = np.clip(M, lower, upper)

# Z-score method to find outlier rows
means = np.nanmean(M, axis=0)
stds = np.nanstd(M, axis=0)
z = (M - means) / stds
outlier_mask = np.abs(z) > 3
rows_with_outliers = np.any(outlier_mask, axis=1)
```

Choose clipping when you want to keep dataset size; use removal when outliers indicate errors.

7. Normalization & scaling

```
# Min-max scaling per column
mins = M.min(axis=0)
maxs = M.max(axis=0)
M_scaled = (M - mins) / (maxs - mins)

# Standard scaling (z-score)
```

```
M_standard = (M - M.mean(axis=0)) / M.std(axis=0)
```

For ML pipelines prefer scikit-learn scalers (they implement fit/transform), but NumPy patterns are the underlying math.

8. Vectorized string ops with `np.char`

```
# Basic string cleanup
s = np.array([" Alice ", "BOB", "cHarlie"])
clean = np.char.strip(s)
clean = np.char.title(clean)
clean = np.char.replace(clean, "\u2019", "'")
```

String operations are available but limited. For heavy text cleaning, use Pandas or dedicated text libraries.

9. Reshaping, stacking & concatenation

```
# Reshape
v = np.arange(12)
M = v.reshape(3,4)

# Stack/concatenate
A = np.random.rand(5,3)
B = np.random.rand(7,3)
full = np.vstack([A, B])  # same number of columns
```

10. Unique values & deduplication

```
# Unique rows
rows = np.array([[1,2], [1,2], [3,4]])
# To get unique rows, view as structured type
uniq_rows = np.unique(rows, axis=0)

# Unique with return index
```

```
vals, idx = np.unique(rows[:,0], return_index=True)
```

Deduplicating structured numeric data is possible but often easier with Pandas for label-aware tasks.

11. Infinities, numeric stability & safe reductions

```
# Detect inf and NaN
np.isinf(M).any(), np.isnan(M).any()

# Replace inf with nan then use nan-aware reductions
M[np.isinf(M)] = np.nan
col_means = np.nanmean(M, axis=0)
```

Be cautious when converting dtypes (float32 vs float64) — choose precision appropriate to your application to avoid unexpected behavior.

12. Useful helper patterns

```
# Replace multiple sentinels using masks
sentinels = [-999, -1e9]
mask = np.isin(M, sentinels)
M[mask] = np.nan

# Apply a function along rows/columns with np.apply_along_axis
(use sparingly)
def robust_mean(row):
    return np.nanmean(row)
col_means = np.apply_along_axis(robust_mean, 0, M)
```

Recommended NumPy Data Preparation Workflow (practical)

1. **Inspect:** shapes, dtypes, quick stats (`arr.shape`, `arr.dtype`, `arr.mean()`, `arr.std()`).
2. **Normalize/standardize:** center and scale numeric features appropriately for models.
3. **Handle missingness:** use masks, replace sentinels, decide between imputation and removal.
4. **Detect & treat outliers:** using percentiles or z-scores.
5. **Ensure correct dtype:** floats for NaN, integers where safe, convert strings explicitly.
6. **Reshape/align:** stack, transpose, or reshape to produce (n_samples, n_features).
7. **Validate:** run sanity checks (no NaN in final feature matrix unless supported by downstream model).
8. **Persist:** save arrays with `np.save` / `np.savez_compressed` or use Pandas for CSVs when human readability is required.

Minimal reproducible NumPy cleaning script (end-to-end)

```
import numpy as np

# Load numeric CSV into NumPy (fast) — use genfromtxt for missing values
data = np.genfromtxt('raw_numeric.csv', delimiter=',', skip_header=1, filling_values=np.nan)

# Basic cleanup
# 1. Remove rows that are all NaN
row_mask = ~np.isnan(data).all(axis=1)
data = data[row_mask]

# 2. Replace sentinel -999 with NaN
data[np.isclose(data, -999)] = np.nan

# 3. Impute column medians
col_medians = np.nanmedian(data, axis=0)
inds = np.where(np.isnan(data))
data[inds] = np.take(col_medians, inds[1])

# 4. Standardize
means = data.mean(axis=0)
stds = data.std(axis=0)
X = (data - means) / stds

# Save
np.savez_compressed('cleaned_data.npz', X=X)
```

Resources & further reading

- Official NumPy documentation — user guide and reference for ndarray and ufuncs.
- "Python Data Science Handbook" by Jake VanderPlas — practical NumPy & Pandas recipes.
- scikit-learn preprocessing docs — for production-ready scalers & transformers.
- Great Expectations / data validation tools — to add checks after cleaning.

If you'd like, I can also add a short section mapping these NumPy patterns to equivalent scikit-learn preprocessing steps or provide downloadable Jupyter notebook examples.