15. NumPy Missing Values Handling

1. Topic Overview

What this topic is

Missing values handling means how NumPy represents and works with data that is not present.

Why it exists

Real data is incomplete.
Sensors fail.
Forms are not filled.
APIs return nulls.

One real-world analogy

A spreadsheet where some cells are empty.
You still must calculate totals.

2. Core Theory (Deep but Clear)

How NumPy thinks about missing data

NumPy arrays are:

- Fixed size
- Fixed dtype
- Stored in contiguous memory

Because of this:

- NumPy **cannot store true nulls for numbers**

- It uses **special values** instead

Missing Value Types in NumPy

1. `np.nan` (Not a Number)

- Used for floating-point arrays
- Represents missing or invalid numbers
- Stored as a special IEEE floating value

Important:

- Only works with float dtype
- Breaks normal comparisons

2. `np.inf` and `-np.inf`

- Represents overflow or division errors
- Not missing, but often treated as invalid

3. `None`

- Python object
- Forces dtype = object
- Very slow
- Not recommended for numeric work

4. Masked values (Advanced concept)

- Uses a separate boolean mask
- Data stays intact
- Mask tells which values are missing

NumPy tool:

- `np.ma` module

3. Syntax & Examples

A. `np.nan`

Basic syntax

```
import numpy as np

x = np.array([1.0, 2.0, np.nan, 4.0])
print(x)
```

Output

```
[ 1.  2. nan  4.]
```

Explanation

- Array dtype is float
- `nan` occupies memory like a float
- Value exists but means missing

Checking for `nan`

```
np.isnan(x)
```

Output

```
[False False  True False]
```

Explanation

- Direct comparison `x == np.nan` fails
- `np.isnan` is the only correct way

Removing `nan`

```
x[~np.isnan(x)]
```

Output

```
[1. 2. 4.]
```

B. `np.inf`

Example

```
y = np.array([1, 2, 0], dtype=float)
z = 10 / y
print(z)
```

Output

```
[10.  5. inf]
```

Explanation

- Division by zero creates `inf`
- Value is valid float but meaningless for ML

Detecting `inf`

```
np.isinf(z)
```

Output

```
[False False  True]
```

C. None

Example

```
a = np.array([1, None, 3])
print(a)
print(a.dtype)
```

Output

```
[1 None 3]
object
```

Explanation

- NumPy switches to object dtype
- No vectorization
- Very slow operations

D. Masked Arrays

Example

```
m = np.ma.array([1, 2, 3], mask=[0, 1, 0])
print(m)
```

Output

```
[1 -- 3]
```

Explanation

- `--` means masked
- Data still exists internally
- Mask controls visibility

4. Why This Matters in Data Science

Data cleaning

- Detect missing sensor readings
- Remove invalid rows
- Replace missing values

Feature engineering

- Mean or median filling
- Indicator features for missingness

Model input preparation

- ML models do not accept `nan`
- Shapes break if rows removed wrongly

ML / DL pipelines

- Training crashes on `nan`
- Loss becomes `nan`
- Gradients explode

What breaks if you don’t understand this

- Wrong statistics
- Silent bugs
- Models that never converge

5. Common Mistakes (VERY IMPORTANT)

1. Using `x == np.nan`
   - Why: misunderstanding float behavior
   - Fix: always use `np.isnan`
2. Mixing `None` and numbers
   - Why: Python habit
   - Fix: use `np.nan` only
3. Forgetting dtype conversion
   - Why: reading CSV data
   - Fix: enforce float dtype
4. Ignoring `inf`
   - Why: not visible like `nan`
   - Fix: check `np.isinf`
5. Removing rows blindly
   - Why: quick cleaning
   - Fix: track labels and shapes

6. Performance & Best Practices

When it is fast

- Float arrays with `np.nan`
- Vectorized masking

When it is slow

- Object dtype arrays
- Python loops for cleaning

Warnings

- `nan` propagates in calculations
- Mean of array with `nan` becomes `nan`
- Use `np.nanmean`, `np.nansum`

7. Practice Problems (NO SOLUTIONS)

Easy (5)

1. Create a float array with one missing value.
2. Detect missing values in a 1D array.
3. Count total missing values.
4. Remove missing values from an array.
5. Replace missing values with zero.

Medium (7)

6. Given sensor data, remove rows with any missing value.
7. Replace missing values with column mean.
8. Detect both `nan` and `inf`.
9. Create a mask for invalid values.
10. Keep shape same while replacing missing values.
11. Count missing values per column.
12. Convert object array with `None` to float.

Hard (5)

13. Normalize data while ignoring missing values.
14. Compute mean per feature with missing data.
15. Build a missing-value indicator feature.
16. Compare results with and without missing handling.
17. Simulate missing data in a clean dataset.

Industry-Level Tasks (3)

18. Clean user activity logs with missing durations.
19. Prepare ML-ready matrix from raw CSV with missing values.
20. Build preprocessing step that safely handles missing data.

8. Mini Checklist

- `np.nan` is only for floats
- Never compare `nan` directly
- Avoid `None` in NumPy arrays
- Always check `inf`
- Missing values break ML models
- Shape consistency matters
- Use NumPy functions, not loops

NumPy Missing Values Handling (Advanced but Mandatory)

1. Topic Overview

What this topic is

These are **advanced but essential NumPy tools** for working safely with missing or invalid data.

Why it exists

Basic `np.nan` handling is not enough for:

- statistics
- preprocessing
- stable ML pipelines

One real-world analogy

You do not just remove empty cells.
You also decide **how calculations should ignore them**.

2. Core Theory (Deep but Clear)

A. NaN-Safe Statistical Functions

Why normal functions fail

NumPy math follows strict rules:

- Any operation with `nan` → result becomes `nan`

Example logic:

```
mean([1, 2, nan]) → nan
```

This breaks analytics.

NumPy solution

NaN-aware functions:

- `np.nanmean`
- `np.nansum`
- `np.nanmin`
- `np.nanmax`
- `np.nanstd`
- `np.nanvar`

Internal behavior:

- NumPy scans array
- Skips `nan` values
- Keeps dtype and shape logic intact

B. `np.nan_to_num`

What it does

Replaces:

- `nan`
- `+inf`
- `-inf`

With safe numeric values.

Why it exists:

- ML models cannot accept invalid floats

C. Boolean Masking for Missing Logic

Core idea

Missing handling is **mask logic**, not deletion.

Mask:

- Boolean array
- Same shape as data
- Controls what is valid

NumPy thinks like this:

```
data + mask → filtered computation
```

D. Axis-Aware Missing Handling

Why axis matters

Real data is 2D or higher.

Examples:

- Rows = samples
- Columns = features

Wrong axis = wrong result.

Axis meaning:

- `axis=0` → column-wise
- `axis=1` → row-wise

E. Shape-Preserving Cleaning

Critical rule

In ML:

- Input shape must remain stable

Deleting values breaks:

- feature alignment
- labels
- batching

Correct approach:

- Replace
- Mask
- Aggregate safely

3. Syntax & Examples

A. NaN-Safe Statistics

Syntax

```
np.nanmean(arr)
```

Example

```
import numpy as np

x = np.array([1, 2, np.nan, 4])
print(np.nanmean(x))
```

Output

```
2.3333333333333335
```

Explanation

- `nan` ignored
- Mean computed from remaining values

Column-wise example

```
X = np.array([
    [1, 2, np.nan],
    [4, np.nan, 6]
])

print(np.nanmean(X, axis=0))
```

Output

```
[2.5 2.  6. ]
```

Explanation:

- Each column handled independently
- Shape preserved

B. `np.nan_to_num`

Syntax

```
np.nan_to_num(arr, nan=0.0)
```

Example

```
x = np.array([1, np.nan, np.inf, -np.inf])
print(np.nan_to_num(x))
```

Output

```
[ 1.00000000e+000  0.00000000e+000  1.79769313e+308
 -1.79769313e+308]
```

Explanation

- `nan` → 0
- `inf` → max float
- Prevents crashes

C. Boolean Masking Logic

Example

```
x = np.array([1, 2, np.nan, 4])
mask = ~np.isnan(x)

print(mask)
print(x[mask])
```

Output

```
[ True  True False  True]
[1. 2. 4.]
```

Explanation

- Mask controls validity
- No mutation of original array

D. Axis-Based Filtering

Example

```
X = np.array([
    [1, 2, np.nan],
    [3, 4, 5]
])

valid_rows = ~np.isnan(X).any(axis=1)
print(valid_rows)
```

Output

```
[False  True]
```

Explanation:

- Row 0 has missing
- Row 1 is clean

E. Shape-Safe Replacement

Example

```
X = np.array([
    [1, np.nan],
    [3, 4]
])

X[np.isnan(X)] = 0
print(X)
```

Output

```
[[1. 0.]
 [3. 4.]]
```

Explanation:

- Shape preserved
- ML-safe

4. Why This Matters in Data Science

Data cleaning

- Safe statistics
- No silent `nan` pollution

Feature engineering

- Mean/variance with missing data
- Stable transformations

Model input preparation

- Fixed input shape
- No runtime crashes

ML / DL pipelines

- Prevent exploding loss
- Avoid invalid gradients

What breaks if ignored

- Wrong feature scaling
- Broken batches
- Models that output `nan`

5. Common Mistakes (VERY IMPORTANT)

1. Using `mean()` instead of `nanmean()`
2. Replacing values before understanding axis
3. Deleting rows without syncing labels
4. Forgetting `inf` handling
5. Breaking shape consistency

6. Performance & Best Practices

Fast

- Vectorized nan functions
- Boolean masking

Slow

- Python loops
- Object arrays

Warnings

- `nan_to_num` can hide data problems
- Always log replacements
- Prefer masking over deletion

7. Practice Problems (NO SOLUTIONS)

Easy (5)

1. Compute mean ignoring missing values.
2. Replace all invalid values with zero.
3. Count missing values per column.
4. Create a validity mask.
5. Detect rows with any missing data.

Medium (7)

6. Normalize data with missing values.
7. Compute variance per feature safely.
8. Replace missing values using column mean.
9. Build feature validity matrix.
10. Preserve shape after cleaning.

11. Detect invalid rows using axis logic.
12. Combine `nan` and `inf` handling.

Hard (5)

13. Implement z-score ignoring missing data.
14. Compare masked vs replaced results.
15. Simulate missing values randomly.
16. Track missing percentage per feature.
17. Prepare batch-safe input tensor.

Industry-Level Tasks (3)

18. Clean IoT sensor streams with gaps.
19. Build preprocessing step for ML inference.
20. Validate model input before prediction.

8. Mini Checklist

- Normal math fails with `nan`
- Use nan-safe NumPy functions
- Masking > deletion
- Axis matters
- Shape stability is mandatory
- ML models reject invalid floats

This completes **Missing Values Handling** properly.