NumPy & Pandas

Developer Handbook

A complete reference guide covering every major function, method, and concept in NumPy and Pandas — with clear descriptions, practical code examples, tips, and best practices for data scientists and Python developers.

NumPy 1.26+ | Pandas 2.x | Python 3.10+ | 300+ Functions

Table of Contents

Introduction to NumPy & Pandas

1.1 What is NumPy?

NumPy (Numerical Python) is the cornerstone library for scientific computing in Python. It provides the **ndarray** — a fast, memory-efficient multi-dimensional array — along with hundreds of mathematical, statistical, and linear-algebra functions. NumPy operations are implemented in C, making them dramatically faster than pure-Python loops.

> ■ *Tip: NumPy arrays are homogeneous (all elements share one dtype). This is key to their speed.*

1.2 What is Pandas?

Pandas builds on NumPy to offer two high-level data structures: **Series** (1-D labeled array) and **DataFrame** (2-D labeled table). It excels at data wrangling, I/O, time-series analysis, and exploratory data analysis — essentially the go-to tool for data science workflows.

1.3 Installation & Import Conventions

```
# Install
pip install numpy pandas

# Standard import aliases used throughout this handbook
import numpy as np
import pandas as pd

# Verify versions
print(np.__version__) # e.g. 1.26.4
print(pd.__version__) # e.g. 2.2.1
```

2.1 From Python Data

np.array(object, dtype=None)

Convert a list / tuple / nested list into an ndarray.

```
a = np.array([1, 2, 3]) # 1-D
b = np.array([[1,2],[3,4]]) # 2-D
c = np.array([1.0, 2, 3], dtype=np.float32)
```

np.asarray(a, dtype=None)

Like np.array but does NOT copy if input is already an ndarray.

```
x = [1, 2, 3]
a = np.asarray(x) # no copy if x is already ndarray
```

np.fromiter(iterable, dtype, count=-1)

Build 1-D array from any Python iterable.

```
gen = (x**2 for x in range(5))
a = np.fromiter(gen, dtype=float) # [0. 1. 4. 9. 16.]
```

np.frombuffer(buffer, dtype=float)

Interpret a buffer (bytes) as a 1-D array.

```
raw = b'\x00\x00\x80?\x00\x00\x00@'
a = np.frombuffer(raw, dtype=np.float32) # [1. 2.]
```

2.2 Filled Arrays

np.zeros(shape, dtype=float)

Array filled with 0.0.

```
np.zeros(5) # [0. 0. 0. 0. 0.]
np.zeros((3, 4)) # 3x4 matrix of zeros
```

`np.ones(shape, dtype=float)`

Array filled with 1.0.

```
np.ones((2, 3)) # [[1. 1. 1.],[1. 1. 1.]]
```

`np.full(shape, fill_value)`

Fill with any scalar.

```
np.full((2, 2), 7) # [[7, 7],[7, 7]]
np.full(4, np.pi) # [3.14159... x4]
```

`np.empty(shape, dtype=float)`

Allocate array WITHOUT initialising values (fast, but values are garbage — always write before read).

```
a = np.empty((3, 3))
a[:] = 0 # fill manually
```

`np.zeros_like(a) / np.ones_like(a) / np.full_like(a, v)`

Create an array of same shape & dtype as `a`, filled with 0, 1, or v.

```
x = np.array([[1, 2], [3, 4]])
np.zeros_like(x) # [[0, 0],[0, 0]]
np.full_like(x, 9) # [[9, 9],[9, 9]]
```

2.3 Range & Sequence Arrays

`np.arange([start,] stop[, step], dtype=None)`

Like Python range() but returns an ndarray. Supports float steps.

```
np.arange(10) # [0..9]
np.arange(1, 10, 2) # [1, 3, 5, 7, 9]
np.arange(0, 1, 0.25) # [0. 0.25 0.5 0.75]
```

`np.linspace(start, stop, num=50, endpoint=True)`

num evenly-spaced values between start and stop (inclusive by default).

```
np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1. ]
np.linspace(0, 1, 5, endpoint=False) # [0. 0.2 0.4 0.6 0.8]
```

`np.logspace(start, stop, num=50, base=10)`

num values evenly spaced on a log scale.

```
np.logspace(0, 3, 4) # [ 1. 10. 100. 1000.]
```

np.geomspace(start, stop, num=50)

Like linspace but the spacing is geometric (ratio between terms is constant).

```
np.geomspace(1, 1000, 4) # [ 1. 10. 100. 1000.]
```

2.4 Matrix & Special Arrays

np.eye(N, M=None, k=0, dtype=float)

Identity matrix (or shifted diagonal). k=0 main diagonal, k>0 above, k<0 below.

```
np.eye(3) # 3x3 identity
np.eye(3, k=1) # ones on super-diagonal
```

np.identity(n, dtype=float)

Strictly square identity matrix — shortcut for np.eye(n).

```
np.identity(4) # 4x4 identity
```

np.diag(v, k=0)

If v is 1-D: construct diagonal matrix. If v is 2-D: extract diagonal.

```
np.diag([1,2,3]) # diagonal matrix
np.diag(np.eye(3)) # [1., 1., 1.]
```

np.tri(N, M=None, k=0, dtype=float)

Lower-triangular matrix of ones.

```
np.tri(4) # 4x4 lower triangular
```

np.tril(m, k=0) / np.triu(m, k=0)

Return lower/upper triangle of matrix; zero out the rest.

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
np.tril(A) # zeros above diagonal
np.triu(A) # zeros below diagonal
```

2.5 Random Arrays (np.random)

> ■ *Tip: Always seed for reproducibility: np.random.seed(42) or rng = np.random.default_rng(42) (recommended new API).*

np.random.rand(d0, d1, …)

Uniform random values in [0, 1). Old API.

```
np.random.rand(3) # 3 values
np.random.rand(2, 4) # 2x4 matrix
```

np.random.randn(d0, d1, …)

Standard normal (mean=0, std=1) samples.

```
np.random.randn(1000) # 1000 normal samples
```

np.random.randint(low, high=None, size=None)

Random integers from low (inclusive) to high (exclusive).

```
np.random.randint(0, 10, size=5) # e.g. [3,7,2,9,0]
```

np.random.choice(a, size=None, replace=True, p=None)

Random sample from array a. replace=False for without-replacement sampling.

```
np.random.choice([10,20,30,40], size=3)
np.random.choice(100, 10, replace=False)
```

np.random.shuffle(x)

Shuffle array x **in place** along first axis.

```
a = np.arange(10)
np.random.shuffle(a) # a is modified in place
```

rng.integers / rng.random / rng.standard_normal

New Generator API (recommended). Create with np.random.default_rng(seed).

```
rng = np.random.default_rng(42)
rng.integers(0, 100, size=5)
rng.random((3, 3))
rng.standard_normal(1000)
```

3.1 Shape & Reshaping

a.shape / a.ndim / a.size / a.dtype

Core attributes: shape tuple, number of dimensions, total elements, data type.

```
a = np.ones((3,4,5))
a.shape # (3, 4, 5)
a.ndim # 3
a.size # 60
a.dtype # float64
```

np.reshape(a, newshape) / a.reshape(newshape)

Return view with new shape (total elements must match). Use -1 to infer one dimension.

```
a = np.arange(12)
a.reshape(3, 4) # 3x4
a.reshape(2, -1) # 2x6, -1 auto-inferred
```

a.ravel() / np.ravel(a)

Return 1-D contiguous flattened view (copy only if necessary).

```
a = np.array([[1,2],[3,4]])
a.ravel() # [1, 2, 3, 4]
```

a.flatten(order='C')

Return 1-D **copy** (always a new array). order: 'C'=row-major, 'F'=column-major.

```
a.flatten() # [1, 2, 3, 4] (copy)
```

np.squeeze(a, axis=None)

Remove axes of length 1.

```
a = np.zeros((1, 3, 1))
np.squeeze(a).shape # (3,)
```

np.expand_dims(a, axis)

Insert a new axis (opposite of squeeze).

```
a = np.array([1,2,3]) # shape (3,)
np.expand_dims(a, 0).shape # (1, 3)
```

3.2 Transposition & Axis Operations

a.T / np.transpose(a, axes=None)

Transpose (reverse axis order, or specify permutation with axes).

```
a = np.arange(6).reshape(2,3)
a.T.shape # (3, 2)
```

np.swapaxes(a, axis1, axis2)

Swap two specific axes.

```
a = np.zeros((2,3,4))
np.swapaxes(a, 0, 2).shape # (4, 3, 2)
```

np.moveaxis(a, source, destination)

Move one or more axes to new positions.

```
a = np.zeros((3,4,5))
np.moveaxis(a, 0, -1).shape # (4, 5, 3)
```

3.3 Joining & Splitting

np.concatenate((a1,a2,...), axis=0)

Join arrays along an existing axis.

```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6]])
np.concatenate((a, b), axis=0) # (3,2)
```

np.vstack(tup) / np.hstack(tup)

Stack row-wise / column-wise.

```
np.vstack([np.array([1,2]), np.array([3,4])])
np.hstack([np.array([[1],[2]]), np.array([[3],[4]])])
```

np.stack(arrays, axis=0)

Join along a **new** axis (all input arrays must have the same shape).

```
a = np.array([1,2,3])
b = np.array([4,5,6])
np.stack([a, b], axis=0).shape # (2, 3)
```

np.split(a, indices_or_sections, axis=0)

Split into multiple sub-arrays.

```
a = np.arange(9)
np.split(a, 3) # three arrays of 3
np.split(a, [3, 6]) # [0:3], [3:6], [6:]
```

3.4 Repeating & Tiling

np.repeat(a, repeats, axis=None)

Repeat each element repeats times.

```
np.repeat([1,2,3], 3) # [1,1,1,2,2,2,3,3,3]
```

np.tile(A, reps)

Tile (stack copies of) array A.

```
np.tile([1,2,3], 3) # [1,2,3,1,2,3,1,2,3]
```

np.pad(array, pad_width, mode='constant')

Pad array with values around its edges.

```
np.pad([1,2,3], 2, mode='constant', constant_values=0)
# [0, 0, 1, 2, 3, 0, 0]
```

4.1 Element-wise Math Functions

> ■ *Tip: All ufuncs support an `out` parameter to write results to an existing array, saving memory.*

np.add / np.subtract / np.multiply / np.divide

Element-wise +, -, *, /. Also available as operators.

```
np.add(a, b) # same as a + b
np.multiply(a, 3) # same as a * 3
```

np.power(x1, x2) / np.sqrt(x) / np.cbrt(x)

Exponentiation, square root, cube root.

```
np.power(2, [1,2,3,4]) # [2, 4, 8, 16]
np.sqrt([1, 4, 9, 16]) # [1. 2. 3. 4.]
```

np.abs(x) / np.fabs(x)

Absolute value.

```
np.abs([-3, 1, -4, 1, 5]) # [3, 1, 4, 1, 5]
```

np.floor / np.ceil / np.round / np.trunc

Rounding operations.

```
np.floor([1.7, -1.7]) # [ 1. -2.]
np.ceil( [1.2, -1.2]) # [ 2. -1.]
np.round([0.5, 1.5, 2.5]) # [0. 2. 2.]
```

np.exp(x) / np.exp2(x) / np.expm1(x)

e^x, 2^x, and e^x - 1.

```
np.exp([0, 1, 2]) # [1. 2.718 7.389]
```

LearnStack Free Preview

This was a free preview. Get the full book on LearnStack.

Visit: https://www.learnstack.co.in

Digital PDF delivery is handled through Gumroad email after purchase.