The NumPy Manual: A Comprehensive Guide to Syntax, Functions, and Application

Generated by Gemini

November 9, 2025

Contents

Preface: A Note on This Document

Purpose of This Manual

This document is a bespoke, comprehensive technical manual designed to consolidate the core concepts, functions, and practical applications of the NumPy library. The user's request for a single, comprehensive "PDF" with "proper examples" for "each function" highlights a common need among developers and researchers: a single, integrated resource that bridges the gap between a conceptual "getting started" guide and an exhaustive, and often overwhelming, API reference.

Addressing Official Documentation

The NumPy project provides its own excellent, official documentation, which is the primary source of truth for the library. These resources are typically, and logically, split into two main components:

1. **The NumPy User Guide:** This document provides in-depth information on the key concepts, background, and explanations of *why* NumPy works the way it does.[2, 3] It is designed to be read like a book, covering topics such as array fundamentals, broadcasting, and indexing.

2. **The NumPy API Reference:** This is an exhaustive, alphabetical, and module-specific listing of every function, object, and module in the library.[2, 5] It details parameters and return values but assumes the reader already understands the core concepts.[2]

While the official project provides these as separate PDF downloads [1, 6], a practitioner often needs a single document that seamlessly integrates these two. A developer may read about the *concept* of broadcasting in the User Guide [3], but they must then hunt through the Reference Guide to find the specific functions that *use* it.

This manual serves that purpose. It is structured as a single, pedagogical, book-length guide. It merges the conceptual explanations of the User Guide with the practical, code-first examples and syntax of the API Reference, focusing on the most critical and foundational components of the library.

Why NumPy? The Lingua Franca of Data

Before diving into the syntax, it is essential to understand *why* this library merits such a detailed study. NumPy (Numerical Python) is, by definition, the "fundamental package for scientific computing in Python".[2, 4, 7] Its value is not just in its own functions but in its role as the bedrock of the entire scientific Python and data science ecosystem.

NumPy's core contribution is the `ndarray` object, a high-performance, multidimensional array.[2, 4] This data structure has become the "lingua franca" for data exchange among nearly all other major data-centric libraries.[8] When you use:

- **Pandas:** The `DataFrame` object is built on top of NumPy arrays to store its data.[9, 10, 11]
- **SciPy:** This library for scientific algorithms is built directly upon NumPy's array object.[10, 12]
- **Matplotlib:** The primary plotting library expects NumPy arrays as the input for most of its functions.[10, 13]
- **Scikit-learn:** This machine learning library uses NumPy arrays for its datasets, models, and results.[9]

Therefore, mastering NumPy is not an optional or specialized skill; it is the prerequisite for effective, high-performance work in data analysis, machine learning, and scientific research in Python.

Chapter 1

The NumPy Foundation: The `ndarray`

1.1 Defining the N-Dimensional Array (`ndarray`)

At the absolute core of the NumPy package is the `ndarray` object.[4] This object encapsulates n-dimensional arrays of homogeneous data types.[4] It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers.[14]

In NumPy, dimensions are referred to as **axes**.[14] For example, a 1D array, or vector, has one axis. A 2D array, or matrix, has two axes. This forms the foundation of all operations.

The `ndarray` is defined by several key properties:

- **Homogeneous:** All elements in a NumPy array are *required* to be of the same data type (e.g., all 64-bit integers or all 64-bit floats). This is a critical distinction from standard Python lists.[4]
- **Fixed Size:** The size of an `ndarray` is established at its creation and cannot be changed dynamically. Changing the size of an array (e.g., by appending) will always create a *new* array and delete the original.[4]
- **N-Dimensional:** An array can have any number of dimensions, from a 1D vector to a 2D matrix to 3D (e.g., a set of images) or higher-dimensional tensors.

1.2 The `ndarray` vs. The Python `list`

For new users, the most common point of confusion is why an `ndarray` is necessary when Python already has a perfectly good `list` container.

A standard Python `list` is an excellent, general-purpose container. It is heterogeneous, meaning it can store elements of different types (e.g., `[1, "hello", 3.14]`), and it is dynamic, allowing you to append or remove elements efficiently.[15]

The `ndarray` trades this flexibility for performance. The rigid constraints of homogeneity and fixed size are deliberate design choices. Because all elements are the same type and the size is known, NumPy can:

1. Store the data in a **contiguous block of memory**, which is highly efficient for CPU access.
2. Perform operations using **compiled C code** instead of interpreted Python.[10]
3. Execute "vectorized" operations that apply a function to all elements simultaneously, rather than iterating through a `for` loop (e.g., `array_a + array_b`).[16, 17]

This vectorized approach is the source of NumPy's speed. For large datasets, operations on an `ndarray` are orders of magnitude faster than equivalent operations on a Python `list`.

1.3 Understanding NumPy Data Types (`dtype`)

Because arrays are homogeneous, the `dtype` (data type) of an array is a critical attribute.[18] This `dtype` object describes the fixed size of the data (e.g., in bytes) and its interpretation.

When you create an array, NumPy attempts to infer the `dtype`. However, you can (and often should) specify it explicitly. The most common `dtypes` include:

- `np.int64`: 64-bit integer.
- `np.float64`: 64-bit floating-point number. This is the default `dtype` for most array creation functions.[13]
- `np.bool_`: Boolean type (True or False), used heavily in masking.
- `np.string_`: Fixed-width ASCII string.[3]
- `np.unicode_`: Fixed-width Unicode string.[3]

You can explicitly set the `dtype` during creation, which is vital for memory management, such as when you know your data will fit into a smaller type.

Example:

```
>>> import numpy as np
>>> # NumPy infers float64
>>> arr_float = np.array([1.0, 2.0, 3.0])
>>> print(arr_float.dtype)
float64

>>> # Explicitly set dtype to 16-bit integer
>>> arr_int = np.ones((2, 3), dtype=np.int16)
>>> print(arr_int.dtype)
int16
```

1.4 Array Creation: From Data and Structures

You can create `ndarrays` in several ways. These functions are the primary entry point into the library.[8, 19]

1.4.1 From Existing Data

The most common method is converting an existing Python list or tuple into an array.

- `np.array(object, dtype=None)`: Converts a Python list, tuple, or other array-like object into an `ndarray`.

Example:

```
>>> # From a 1D Python list
>>> a = np.array([1, 2, 3])
>>> print(a)
[1 2 3]

>>> # From a 2D Python list (list of lists)
>>> b = np.array([(1.5, 2, 3), (4, 5, 6)])
```

```
>>> print(b)
[[1.5 2.  3. ]
 [4.  5.  6. ]]

>>> # From a Python tuple
>>> c = np.array((5, 6, 7, 8))
>>> print(c)
[5 6 7 8]
```

1.4.2 Placeholder Arrays

These functions are used when you know the desired *shape* of the array but not yet its contents. They are highly efficient as they can pre-allocate the memory.

- `np.zeros(shape)`: Creates an array full of zeros.[13, 19]
- `np.ones(shape)`: Creates an array full of ones.[13, 19]
- `np.empty(shape)`: Creates an array whose initial content is ”random”.[13] This function is faster than `np.zeros` because it does not initialize the values. It should be used only when you intend to immediately overwrite every element.
- `np.full(shape, fill_value)`: Creates an array of a given shape, filling it with a constant value.[19]

Example:

```
>>> # A 3x4 array of zeros
>>> z = np.zeros((3, 4))
>>> print(z)
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

>>> # A 2x2 constant array
>>> f = np.full((2, 2), 7)
>>> print(f)
[[7 7]
 [7 7]]

>>> # An uninitialized 2x2 array
>>> e = np.empty((2, 2))
>>> print(e)  # Output will vary based on memory state
[[6.95156579e-310 1.39031316e-309]
 [1.39031316e-309 1.39031316e-309]]
```

1.4.3 Sequence-Based Arrays

These functions create arrays based on numerical sequences.

- `np.arange(start, stop, step)`: Like Python’s built-in `range` function, but returns an `ndarray` instead of a list.[19]
- `np.linspace(start, stop, num)`: Creates an array of `num` evenly spaced values between `start` and `stop` (inclusive).[19] This is one of the most useful functions in scientific computing. Note the final argument is the *number of samples*, not the step size.

Example:

```
>>> # An array of values from 10 to 24, with a step of 5
>>> d = np.arange(10, 25, 5)
>>> print(d)
[10 15 20]

>>> # An array of 9 evenly spaced values from 0 to 2
>>> l = np.linspace(0, 2, 9)
>>> print(l)
[0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
```

1.4.4 Matrix-Specific Arrays

- `np.eye(N)`: Creates a square $N \times N$ identity matrix (ones on the diagonal, zeros elsewhere).[19]
- `np.diag(v)`: If `v` is a 1D array, creates a 2D array with `v` on the diagonal. If `v` is a 2D array, extracts the diagonal as a 1D array.

Example:

```
>>> # A 3x3 identity matrix
>>> i = np.eye(3)
>>> print(i)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Quick Reference: Array Creation Functions

Table 1.1: Quick Reference: Array Creation Functions

Function	Description	Example
`np.array(data)`	Create array from list/tuple	`np.array([1,2,3])`
`np.zeros(shape)`	Create array of zeros [19]	`np.zeros((2, 3))`
`np.ones(shape)`	Create array of ones [19]	`np.ones((2, 3))`
`np.empty(shape)`	Create uninitialized array [19]	`np.empty((2, 3))`
`np.full(shape, val)`	Create array of `val` [19]	`np.full((2, 2), 7)`
`np.arange(start, stop, step)`	Create range array [19]	`np.arange(0, 10, 2)`
`np.linspace(start, stop, num)`	Create `num` points [19]	`np.linspace(0, 1, 5)`
`np.eye(N)`	Create NxN identity matrix [19]	`np.eye(3)`

Chapter 2

Array Mechanics: Accessing and Shaping Data

Once an array is created, the next step is to access, inspect, and manipulate its data and structure.

2.1 Inspecting Array Properties

These attributes allow you to understand the size and type of your `ndarray`.

- `arr.shape`: Returns a tuple of the array's dimensions (e.g., `(rows, columns)`).[14, 18]
- `arr.size`: Returns the total number of elements in the array.[18]
- `arr.ndim`: Returns the number of dimensions (axes).[14]
- `arr.dtype`: Returns the data type of the array's elements.[18]

Example:

```
>>> arr = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(arr.shape)
(2, 3)
>>> print(arr.size)
6
>>> print(arr.ndim)
2
>>> print(arr.dtype)
int64
```

2.2 Indexing and Slicing

NumPy array indexing is a powerful tool that builds on Python's standard slicing.[20]

- **Basic Indexing (1D):** Works just like a Python list. `arr[5]` accesses the sixth element, and `arr[-1]` accesses the last.
- **Basic Slicing (1D):** `arr[start:stop:step]` selects a range of elements.
- **Indexing and Slicing (2D/ND):** This is where NumPy's power becomes clear. You use a comma-separated tuple to index across multiple dimensions: `arr[row, column]`.[20] Slicing works in each dimension.

Example (2D):

```
>>> arr = np.array([[ 1,  2,  3,  4],
...                 [ 5,  6,  7,  8],
...                 [ 9, 10, 11, 12]])
>>> print(arr)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

>>> # Get a single element: 2nd row, 3rd column (0-indexed)
>>> print(arr[1, 2])
7

>>> # Get the first row (all columns)
>>> print(arr[0, :])
[1 2 3 4]

>>> # Get the second column (all rows)
>>> print(arr[:, 1])
[ 2  6 10]

>>> # Get a sub-array: first 2 rows, columns 1 and 2
>>> print(arr[0:2, 1:3])
[[2 3]
 [6 7]]
```

2.3 Advanced Indexing

NumPy provides two more powerful indexing methods that move beyond simple slices.

2.3.1 Integer Array Indexing

You can use arrays of integers to select specific elements. This allows for arbitrary selection and reordering.

Example:

```
>>> arr = np.array([10, 20, 30, 40, 50])
>>> # Select elements at indices 0, 4, and 2
>>> print(arr[[0, 4, 2]])
[10 50 30]
```

2.3.2 Boolean Indexing (Masking)

This is a cornerstone of data analysis. It allows you to select elements from an array based on a *condition*. The process involves two steps:

1. Create a ”mask” array of the same shape as your data, containing `True` or `False` values based on a condition (e.g., `arr > 5`).
2. Pass this mask array into the original array’s index.

This will return a *new* 1D array containing only the elements where the mask was `True`.

Example:

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> print(data)
[[1 2]
 [3 4]
 [5 6]]

>>> # Step 1: Create the mask
>>> mask = (data > 3)
>>> print(mask)
[[False False]
 [False  True]
 [ True  True]]

>>> # Step 2: Apply the mask
>>> print(data[mask])
[4 5 6]

>>> # This can be done in one line:
>>> evens = data[data % 2 == 0]
>>> print(evens)
[2 4 6]
```

2.4 The Critical Distinction: Copies vs. Views

This concept is arguably the single most common source of bugs for new NumPy users. It is essential to understand the distinction between a ”copy” and a ”view”.[3]

In Python, a list slice like `new_list = old_list[0:5]` creates a *copy* of the data. Modifying `new_list` will *not* affect `old_list`.

In NumPy, a **basic slice** (like `arr[0:5]` or `arr[:, 1]`) creates a **view**. A view is a window into the *same* block of memory as the original array. It does not own its data. This is a deliberate performance feature to avoid copying massive arrays, but it has a critical side effect:

If you modify a view, you are also modifying the original array.

Demonstration:

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> # Create a view using a basic slice
>>> a_view = arr[0:3]
>>> print(a_view)
[1 2 3]

>>> # Modify the view
>>> a_view[0] = 99
>>> print(a_view)
[99  2  3]

>>> # Check the original array
>>> print(arr)
[99  2  3  4  5]  # The original array was changed!
```

Conversely, **advanced indexing** (with integer arrays or Boolean masks) creates a **copy** of the data.

How to force a copy: If you want a copy of a slice, you must explicitly use the `np.copy()` function.[21]

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> # Create an explicit COPY
>>> a_copy = arr[0:3].copy()
>>> a_copy[0] = 99
>>> print(a_copy)
[99  2  3]
>>> print(arr)
[1 2 3 4 5]  # The original array is unchanged
```

2.5 Array Manipulation and Reshaping

These functions change the shape or structure of arrays without changing their data.

2.5.1 Changing Shape

- `np.reshape(shape)`: Changes the shape of an array without changing its data. The new shape must be compatible with the original size. You can use `-1` as a wildcard for one dimension, and NumPy will infer its size.
- `.ravel()`: Returns a 1D "flattened" version of the array. This is often a *view*, so it is very fast.
- `.flatten()`: Returns a 1D "flattened" version, but it is always a *copy*.

Example:

```
>>> arr = np.arange(1, 10)  # [1 2 3 4 5 6 7 8 9]
>>> # Reshape to a 3x3 matrix
>>> m = arr.reshape((3, 3))
>>> print(m)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

>>> # Use -1 wildcard to let NumPy calculate the dimension
>>> m2 = arr.reshape((3, -1)) # Same as (3, 3)
```

2.5.2 Combining Arrays

These functions join multiple arrays together.[22]

- `np.concatenate((a, b), axis=0)`: The primary function. Joins arrays along a specified axis.
  - `axis=0`: (Default) Stacks arrays vertically (row-wise).
  - `axis=1`: Stacks arrays horizontally (column-wise).
- `np.vstack((a, b))`: Helper function for `np.concatenate(axis=0)`.[22]
- `np.hstack((a, b))`: Helper function for `np.concatenate(axis=1)`.[22]

Example:

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

>>> # Stack horizontally (axis=1) - requires compatible shapes
>>> c = np.array([[5, 6], [7, 8]])
>>> np.hstack((a, c))
[[1 2 5 6]
 [3 4 7 8]]
```

2.5.3 Splitting Arrays

These functions split a single array into multiple sub-arrays.[22]

- `np.split(arr, indices_or_sections, axis=0)`: The primary function.
- `np.hsplit(arr, 3)`: Helper to split horizontally (into 3 equal sub-arrays).[22]
- `np.vsplit(arr, 2)`: Helper to split vertically (into 2 equal sub-arrays).[22]

2.5.4 Transposing and Axes

- `arr.T`: A simple attribute that returns the transpose (swaps rows and columns) of the array.[22]
- `np.transpose(arr, axes)`: A more general function for N-dimensional arrays that allows you to permute the axes.

Chapter 3

Computation: Ufuncs, Broadcasting, and Statistics

This section covers the core of NumPy's computational power: how it performs mathematical and statistical operations efficiently.

3.1 The Core Paradigm: Universal Functions (ufuncs)

A Universal Function, or **ufunc**, is a function that operates on `ndarrays` in an element-by-element fashion.[16, 23] This is the "vectorized" wrapper that makes NumPy fast. When you call a ufunc, the underlying C-based loop (not a slow Python loop) is executed.[23]

Ufuncs handle array broadcasting and type casting automatically.[16] They come in two primary flavors [17]:

1. **Unary ufuncs:** Operate on a single input array.
2. **Binary ufuncs:** Operate on two input arrays.

3.1.1 Unary Ufuncs

These functions perform an operation on each element of an array.

- `np.sqrt(arr)`: Square root of each element.[24]
- `np.sin(arr)`: Sine of each element.[24]
- `np.log(arr)`: Natural log of each element.[24]
- `np.abs(arr)`: Absolute value of each element.[24]
- `np.ceil(arr)`: Rounds up to the nearest integer.[24]

Example:

```
>>> arr = np.array([1, 4, 9, 16])
>>> print(np.sqrt(arr))
[1. 2. 3. 4.]
```

3.1.2 Binary Ufuncs

These functions perform an element-wise operation on two arrays.

- `np.add(arr1, arr2)`: Element-wise addition.[17]
- `np.subtract(arr1, arr2)`: Element-wise subtraction.[17]
- `np.multiply(arr1, arr2)`: Element-wise multiplication.[17, 24]
- `np.divide(arr1, arr2)`: Element-wise division.[17, 24]
- `np.power(arr1, arr2)`: Element-wise raise `arr1` to the power of `arr2`.[24]
- `np.maximum(arr1, arr2)`: Element-wise maximum.
- `np.array_equal(arr1, arr2)`: Returns `True` if arrays have same shape and elements.[24]

3.1.3 Syntactic Sugar

NumPy's ufuncs are "attached" to the standard Python arithmetic operators. When you use `+`, `-`, `*`, or `/` on two arrays, you are *internally* calling the `np.add`, `np.subtract`, `np.multiply`, or `np.divide` ufunc.[17]

Example:

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])

>>> # These two lines are equivalent
>>> print(np.add(a, b))
[5 7 9]
>>> print(a + b)
[5 7 9]
```

3.2 The Broadcasting Mechanism

Broadcasting is the second-most-powerful (and second-most-confusing) concept in NumPy. It describes how NumPy treats arrays of *different shapes* during arithmetic operations.[3, 23]

Broadcasting allows a ufunc to operate on two arrays of different shapes by "stretching" the smaller array to match the shape of the larger one. This "stretching" is a virtual operation; no data is actually duplicated. Instead, NumPy uses a "stride of 0" on the smaller array's dimension, effectively reusing the same value.[23]

This "magic" follows a strict, explicit algorithm. Two arrays are compatible for broadcasting if they can be made to have the same shape by following these rules:

1. If the two arrays differ in their number of dimensions (e.g., a 2D matrix and a 1D vector), the shape of the one with fewer dimensions is *padded with ones on its left side*.
2. The arrays are then compatible in a dimension if they have the *same size* in that dimension, or if *one* of them has a size of `1`.
3. If these conditions are met, the array with size `1` in any dimension is "broadcast" (or "stretched") to match the other array's size in that dimension.

3.2.1 Example 1: Array and Scalar

- `arr` (shape `(3, 4)`) + `scalar` (value `1`)
- Rule 1: `scalar` shape is `()` -> `(1, 1)`.
- Rule 2: `arr` (3, 4) vs. `scalar` (1, 1) -> `arr` (3, 4) vs. `scalar` (3, 4)
- Result: The `1` is added to every element of `arr`.

3.2.2 Example 2: Array and 1D Vector

- `arr` (shape `(3, 4)`) + `vec` (shape `(4,)`)
- Rule 1: `vec` (4,) -> `(1, 4)`
- Rule 2:
  - Dim 0: `arr` is 3, `vec` is 1. Compatible. `vec` is stretched to 3.
  - Dim 1: `arr` is 4, `vec` is 4. Compatible.
- Result: `vec` (shape `(1, 4)`) becomes `(3, 4)`. The vector `[v1, v2, v3, v4]` is added to *each row* of `arr`.

3.2.3 Example 3: Array and 1D Column

- `arr` (shape `(3, 4)`) + `col` (shape `(3, 1)`)
- Rule 1: No padding needed.
- Rule 2:
  - Dim 0: `arr` is 3, `col` is 3. Compatible.
  - Dim 1: `arr` is 4, `col` is 1. Compatible. `col` is stretched to 4.
- Result: `col` (shape `(3, 1)`) becomes `(3, 4)`. The column vector is added to *each column* of `arr`.

3.3 Mathematical and Arithmetic Operations

Beyond simple ufuncs, NumPy provides functions for more complex matrix operations.

- `np.dot(a, b)`: The dot product. For 2D arrays (matrices), this is matrix multiplication.
- `np.matmul(a, b)` or `a @ b`: The matrix product of two arrays. This is the preferred way to express matrix multiplication.[25] The `@` operator was introduced in Python 3.5 to formalize this.
- `np.inner(a, b)`: The inner product of two arrays.[25]
- `np.outer(a, b)`: The outer product of two vectors.[25]

3.4 Statistical Analysis

NumPy is a powerful tool for descriptive statistics.[16, 18, 21]

- **Aggregate Functions:**
  - `np.sum()`: Sum of all elements.
  - `np.prod()`: Product of all elements.
- **Descriptive Statistics:**
  - `np.mean()`: Arithmetic mean.[16, 21]
  - `np.median()`: Median of all elements.[16]
  - `np.std()`: Standard deviation.[16]
  - `np.var()`: Variance.[16]
- **Positional Statistics:**
  - `np.min()`: Minimum value.[21]
  - `np.max()`: Maximum value.[21]
  - `np.argmin()`: Returns the *index* of the minimum value.
  - `np.argmax()`: Returns the *index* of the maximum value.

3.4.1 The `axis` Parameter: The Key to Data Analysis

All of these statistical functions accept a crucial `axis` parameter. This parameter allows you to perform the operation *along a specified dimension* instead of on the entire array.

- `axis=0`: Performs the operation ”down the columns,” resulting in a 1D array of values for each column.
- `axis=1`: Performs the operation ”across the rows,” resulting in a 1D array of values for each row.

Example:

```
>>> arr = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(arr)
[[1 2 3]
 [4 5 6]]

>>> # Get the sum of all elements
>>> print(arr.sum())
21

>>> # Get the sum "down the columns" (axis=0)
>>> # [1+4, 2+5, 3+6]
>>> print(arr.sum(axis=0))
[5 7 9]

>>> # Get the sum "across the rows" (axis=1)
>>> # [1+2+3, 4+5+6]
>>> print(arr.sum(axis=1))
[6 15]
```

Chapter 4

In-Depth Modules: Linear Algebra and Random Sampling

While the core of NumPy is the `ndarray` object and its basic operations, much of its power for scientific computing comes from its submodules. The two most critical are `numpy.linalg` and `numpy.random`.

4.1 The `numpy.linalg` Module

The `numpy.linalg` module provides the core functions of linear algebra.[25] These functions are implemented as high-performance wrappers around established, battle-tested FORTRAN libraries like LAPACK.

The functions in this module can be logically grouped by their mathematical purpose.[25]

4.1.1 Category 1: Solving Equations and Inverting Matrices

These functions are used to solve linear systems or find the inverse of a matrix.[25]

- `np.linalg.solve(a, b)`: Solves a linear matrix equation $Ax = b$ for the unknown vector x. The matrix A must be square and full-rank (i.e., invertible).[26]
- `np.linalg.inv(a)`: Computes the (multiplicative) inverse of a matrix.[25]
- `np.linalg.pinv(a)`: Computes the (Moore-Penrose) pseudo-inverse. This is used for cases where the matrix is not square or is singular (non-invertible).[25]

Syntax and Example: `np.linalg.solve` To solve the system of equations: $x_0 + 2x_1 = 1$ $3x_0 + 5x_1 = 2$

This can be written in matrix form as $Ax = b$, where: $A = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$

Code:

```
>>> import numpy as np

>>> # Coefficient matrix 'a'
>>> a = np.array([[1, 2], [3, 5]])

>>> # Ordinate or "dependent variable" values 'b'
>>> b = np.array([1, 2])
```

```
>>> # Compute the "exact" solution, x
>>> x = np.linalg.solve(a, b)
>>> print(x)
[-1.  1.]

>>> # This means x0 = -1 and x1 = 1
>>> # Check that the solution is correct:
>>> print(np.allclose(np.dot(a, x), b))
True
```

4.1.2 Category 2: Decompositions

These functions decompose a matrix into its constituent parts, which is fundamental to many advanced algorithms.

- `np.linalg.qr(a)`: Compute the QR factorization of a matrix.[27]
- `np.linalg.svd(a)`: Compute the Singular Value Decomposition (SVD).
- `np.linalg.cholesky(a)`: Compute the Cholesky decomposition (for positive-definite matrices).

4.1.3 Category 3: Matrix Eigenvalues

- `np.linalg.eig(a)`: Compute the eigenvalues and right eigenvectors of a square matrix.[25]
- `np.linalg.eigh(a)`: A specialized, faster version for Hermitian or real symmetric matrices.[25]
- `np.linalg.eigvals(a)`: Computes *only* the eigenvalues of a matrix.[25]

4.1.4 Category 4: Norms and Other Numbers

- `np.linalg.norm(x, ord=None)`: Compute the matrix or vector norm.[25] By default (`ord=None` for a vector), it computes the L2 (Euclidean) norm.
- `np.linalg.det(a)`: Compute the determinant of an array.[25]
- `np.linalg.slogdet(a)`: Compute the sign and (natural) logarithm of the determinant.[25] This is used to prevent numerical underflow or overflow when working with determinants that are very close to zero or extremely large.
- `np.linalg.matrix_rank(A)`: Return the rank of a matrix using the SVD method.[25]

4.2 The `numpy.random` Module

The `numpy.random` module provides functions for generating pseudo-random numbers.

A critical, modern development in the NumPy library is a change in the random number generation (RNG) API.[28]

Older tutorials and codebases (pre-NumPy 1.17) use "legacy" functions like `np.random.rand()`, `np.random.randn()`, and `np.random.randint()`.[29, 30] These functions rely on a global RNG state.

The **correct, modern, and best-practice** method is to first instantiate a `Generator` object using `default_rng()`, and then call methods on that object. This allows for better reproducibility and statistical properties.

4.2.1 The Modern Paradigm: `default_rng`

All modern random number generation should begin with this pattern:

```
>>> from numpy.random import default_rng
>>> # Create a Generator object
>>> rng = default_rng()
```

All subsequent examples will use this `rng` object.

4.2.2 Generating Random Data

- `rng.integers(low, high=None, size=None, endpoint=False)`: The modern function for generating random integers. This is more flexible than the old `np.random.randint`.
  - If `high` is `None`, generates from 0 to `low` (exclusive).
- `rng.random(size=None)`: The modern function for generating random floats in the half-open interval $[0.0, 1.0)$.

Example:

```
>>> # A 2x4 array of integers between 0 and 4 (inclusive)
>>> print(rng.integers(5, size=(2, 4)))
[[4 0 3 0]
 [3 2 2 0]]

>>> # A 1D array of 5 random floats
>>> print(rng.random(5))
[0.7877579  0.01723754 0.93995075 0.17126388 0.69913594]
```

4.2.3 Sampling from Distributions

The `rng` generator can sample from a wide variety of statistical distributions.

- `rng.normal(loc=0.0, scale=1.0, size=None)`: Sample from a normal (Gaussian) distribution. `loc` is the mean, `scale` is the standard deviation. This replaces `np.random.randn()`.
- `rng.uniform(low=0.0, high=1.0, size=None)`: Sample from a uniform distribution.

4.2.4 Sampling and Shuffling

- `rng.choice(a, size=None, replace=True)`: Chooses a random sample from a given 1D array `a`.[29]
- `rng.shuffle(a)`: Modifies a sequence by shuffling its contents *in-place*.
- `rng.permutation(a)`: Returns a *new*, shuffled array. The original array is *not* changed. (This connects back to the "Copies vs. Views" paradigm).

Example:

```
>>> array1 = np.array([1, 2, 3, 4, 5])
>>> # Choose one random number from array1
>>> random_choice = rng.choice(array1)
>>> print(random_choice)
3

>>> # Shuffle array1 in-place
>>> rng.shuffle(array1)
>>> print(array1)
[4 2 5 1 3]

>>> # Get a permuted copy
>>> array2 = np.array([1, 2, 3])
>>> p = rng.permutation(array2)
>>> print(p)
[2 3 1]
>>> print(array2)
[1 2 3]
```

Chapter 5

Persistence and Interoperability

A final, critical component of any library is its ability to interact with the outside world: saving data to disk and working with other libraries.

5.1 I/O with NumPy: Saving and Loading

NumPy provides simple, efficient functions for saving and loading `ndarrays`.[3, 21]

5.1.1 Binary Formats (.npy,.npz)

These are NumPy's native, high-performance, binary formats. They are fast, preserve the `dtype` and `shape` information, and are efficient in terms of storage. They are not, however, human-readable.

- `np.save('filename.npy', arr)`: Save a *single* array to a `.npy` file.[21, 22]
- `np.load('filename.npy')`: Load a single array from a `.npy` file.[21, 22]
- `np.savez('filename.npz', name1=arr1, name2=arr2)`: Save *multiple* arrays into a single, uncompressed `.npz` archive. You access the arrays by the names you provide.[22]
- `np.load('filename.npz')`: Loads an `.npz` file, returning a `NpzFile` object (a dict-like object) from which you can access the arrays.

Example:

```
>>> a = np.array([1, 2, 3])
>>> b = np.arange(4).reshape((2, 2))

>>> # Save a single array
>>> np.save('my_array.npy', a)
>>> # Load it back
>>> loaded_a = np.load('my_array.npy')

>>> # Save multiple arrays
>>> np.savez('my_archive.npz', array_a=a, matrix_b=b)
>>> # Load the archive
>>> archive = np.load('my_archive.npz')
>>> print(archive['array_a'])
[1 2 3]
>>> print(archive['matrix_b'])
[[0 1]
 [2 3]]
```

5.1.2 Text Formats (.txt,.csv)

These functions are used for interoperability with other software (like R, Excel, or gnuplot) that uses plain-text files. They are human-readable but are much slower and larger, and may lose precision or complex `dtype` information.

- `np.savetxt('filename.txt', arr, delimiter=' ')`: Save an array to a text file.[21] Common delimiters include `' '` (space), `','` (CSV), or `'
t'` (TSV).
- `np.loadtxt('filename.txt', delimiter=' ')`: Load data from a text file.[21]

Example:

```
>>> arr = np.array([[1.1, 2.2], [3.3, 4.4]])
>>> # Save as a CSV
>>> np.savetxt('my_data.csv', arr, delimiter=',')
>>> # Load it back
>>> loaded_csv = np.loadtxt('my_data.csv', delimiter=',')
```

5.2 Interoperability: The Bedrock of the PyData Ecosystem

This manual concludes by returning to the premise from the Preface: NumPy is ”fundamental” because it is the ”lingua franca” of the entire scientific Python (PyData) ecosystem.[8] Its `ndarray` object is the standard data container that all other major libraries are designed to accept.

- **NumPy and Pandas:** A `pandas.DataFrame` is, at its core, a collection of NumPy arrays (or extensions thereof). The most common way to create a DataFrame is from a 2D NumPy array or a dictionary of 1D arrays.[10, 11, 31]
- **NumPy and Matplotlib:** Matplotlib, the primary plotting library, is designed to work directly with NumPy arrays. The standard workflow for creating a plot involves generating NumPy arrays and passing them to `matplotlib.pyplot` functions.[10, 13]

```
import matplotlib.pyplot as plt
# x is a NumPy array
x = np.linspace(0, 2 * np.pi, 100)
# y is a NumPy array
y = np.sin(x)
# Matplotlib plots the arrays
plt.plot(x, y)
```

- **NumPy and SciPy:** The SciPy library is built *on top* of NumPy.[10, 12] It provides the more advanced, specialized scientific algorithms that do not belong in the ”fundamental” NumPy package. For example, `scipy.linalg` [32] contains more matrix decompositions than `numpy.linalg`, and `scipy.optimize` provides functions for minimization and regression. SciPy’s functions all take NumPy arrays as their inputs and produce NumPy arrays as their outputs.

Conclusion

This manual has provided a comprehensive, structured overview of the NumPy library. The journey began with the foundational `ndarray` object, highlighting its key properties (homogeneity, fixed size) as deliberate trade-offs for performance. It established the `ndarray`, not the Python `list`, as the only choice for high-speed numerical computation.

The report then detailed the core mechanics of array-based computing: inspecting, indexing, and manipulating array shapes. It placed special emphasis on two critical, and often misunderstood, concepts: the **Copies vs. Views** memory model and the **Broadcasting** mechanism. Mastering these two concepts is the key to moving from a novice to an expert NumPy user.

With the mechanics established, the report explored the *payoff* for this structure: **Universal Functions (ufuncs)**. These vectorized, compiled functions are the source of NumPy's speed. This paradigm was extended to the critical submodules `numpy.linalg` and `numpy.random`, providing a curated, code-first guide to the most essential functions for linear algebra and random sampling, and emphasizing the modern, best-practice `default_rng` generator.

Finally, the report demonstrated NumPy's role as the "lingua franca" [8] of the PyData stack, showing how it is the essential input and output format for Pandas, Matplotlib, and SciPy.[10, 11]

In sum, NumPy is far more than a simple array library. It is the "fundamental package for scientific computing in Python" [2, 4], providing a new computational paradigm based on vectorization and a common data structure that enables an entire ecosystem of advanced tools.