NumPy Cheatsheet

Essential operations for numerical computing and array manipulation

This cheatsheet provides a quick reference to fundamental NumPy operations, syntax, and advanced features, ideal for both beginners and experienced data scientists for efficient numerical computing and array processing.

Array Creation

Create and initialize arrays

Array Indexing

Access and subset array data

Array Operations

Mathematical and logical operations

Statistical Functions

Perform statistical computations

Linear Algebra

Matrix operations and decompositions

Array Creation & Initialization

From Lists: `np.array()`

Create arrays from Python lists or nested lists.

```
import numpy as np
# 1D array from list
arr = np.array([1, 2, 3, 4])
# 2D array from nested lists
arr2d = np.array([[1, 2], [3, 4]])
# Specify data type
arr = np.array([1, 2, 3], dtype=float)
# Array of strings
arr_str = np.array(['a', 'b', 'c'])
```

Zeros and Ones: `np.zeros()` / `np.ones()`

Create arrays filled with zeros or ones.

```
# Array of zeros
zeros = np.zeros(5)  # 1D
zeros2d = np.zeros((3, 4))  # 2D
# Array of ones
ones = np.ones((2, 3))
# Specify data type
zeros_int = np.zeros(5, dtype=int)
```

Identity Matrix: `np.eye()` / `np.identity()`

Create identity matrices for linear algebra operations.

```
# 3x3 identity matrix
identity = np.eye(3)
# Alternative method
identity2 = np.identity(4)
```

Range Arrays: `np.arange()` / `np.linspace()`

Create arrays with evenly spaced values.

```
# Similar to Python range
arr = np.arange(10)  # 0 to 9
arr = np.arange(2, 10, 2)  # 2, 4, 6, 8
# Evenly spaced values
arr = np.linspace(0, 1, 5)  # 5 values from 0 to 1
# Including endpoint
arr = np.linspace(0, 10, 11)
```

Random Arrays: `np.random`

Generate arrays with random values.

```
# Random values between 0 and 1
rand = np.random.random((2, 3))
# Random integers
rand_int = np.random.randint(0, 10, size=(3, 3))
# Normal distribution
normal = np.random.normal(0, 1, size=5)
# Set random seed for reproducibility
np.random.seed(42)
```

Special Arrays: `np.full()` / `np.empty()`

Create arrays with specific values or uninitialized.

```
# Fill with specific value
full_arr = np.full((2, 3), 7)
# Empty array (uninitialized)
empty_arr = np.empty((2, 2))
# Like existing array shape
like_arr = np.zeros_like(arr)
```

Array Properties & Structure

Understand the structure and attributes of your arrays.

01

Basic Properties: `shape` / `size` / `ndim`

Get fundamental information about array dimensions and size.

```
# Array dimensions (tuple)
arr.shape
# Total number of elements
arr.size
# Number of dimensions
arr.ndim
# Data type of elements
arr.dtype
# Size of each element in bytes
arr.itemsize
```

02

Array Info: `arr.info()` / Memory Usage

Get detailed information about array memory usage and structure.

```
# Memory usage in bytes
arr.nbytes
# Array info (for debugging)
arr.flags
# Check if array owns its data
arr.owndata
# Base object (if array is a view)
arr.base
```

03

Data Types: `astype()`

Convert between different data types efficiently.

```
# Convert to different type
arr.astype(float)
arr.astype(int)
arr.astype(str)
# More specific types
arr.astype(np.float32)
arr.astype(np.int16)
```

Array Indexing & Slicing

Basic Indexing: `arr[index]`

Access individual elements and slices.

```
# Single element
arr[0]  # First element
arr[-1]  # Last element
# 2D array indexing
arr2d[0, 1]  # Row 0, Column 1
arr2d[1]  # Entire row 1
# Slicing
arr[1:4]  # Elements 1 to 3
arr[::2]  # Every second element
arr[::-1]  # Reverse array
```

Boolean Indexing: `arr[condition]`

Filter arrays based on conditions.

```
# Simple condition
arr[arr > 5]
# Multiple conditions
arr[(arr > 2) & (arr < 8)]
arr[(arr < 2) | (arr > 8)]
# Boolean array
mask = arr > 3
filtered = arr[mask]
```

Advanced Indexing: Fancy Indexing

Use arrays of indices to access multiple elements.

```
# Index with array of indices
indices = [0, 2, 4]
arr[indices]
# 2D fancy indexing
arr2d[[0, 1], [1, 2]]  # Elements (0,1) and (1,2)
# Combined with slicing
arr2d[1:, [0, 2]]
```

Where Function: `np.where()`

Conditional selection and element replacement.

```
# Find indices where condition is true
indices = np.where(arr > 5)
# Conditional replacement
result = np.where(arr > 5, arr, 0)  # Replace values ≤5 with 0
# Multiple conditions
result = np.where(arr > 5, 'high', 'low')
```

Array Manipulation & Reshaping

Reshaping: `reshape()` / `resize()` / `flatten()`

Change array dimensions while preserving data.

```
# Reshape (creates view if possible)
arr.reshape(2, 3)
arr.reshape(-1, 1)  # -1 means infer dimension
# Resize (modifies original array)
arr.resize((2, 3))
# Flatten to 1D
arr.flatten()  # Returns copy
arr.ravel()  # Returns view if possible
```

Transposing: `T` / `transpose()`

Swap array axes for matrix operations.

```
# Simple transpose
arr2d.T
# Transpose with axes specification
arr.transpose()
np.transpose(arr)
# For higher dimensions
arr3d.transpose(2, 0, 1)
```

Adding/Removing Elements

Modify array size by adding or removing elements.

```
# Append elements
np.append(arr, [4, 5])
# Insert at specific position
np.insert(arr, 1, 99)
# Delete elements
np.delete(arr, [1, 3])
# Repeat elements
np.repeat(arr, 3)
np.tile(arr, 2)
```

Combining Arrays: `concatenate()` / `stack()`

Join multiple arrays together.

```
# Concatenate along existing axis
np.concatenate([arr1, arr2])
np.concatenate([arr1, arr2], axis=1)
# Stack arrays (creates new axis)
np.vstack([arr1, arr2])  # Vertically
np.hstack([arr1, arr2])  # Horizontally
np.dstack([arr1, arr2])  # Depth-wise
```

Mathematical Operations

Basic Arithmetic: `+`, `-`, `*`, `/`

Element-wise arithmetic operations on arrays.

```
# Element-wise operations
arr1 + arr2
arr1 - arr2
arr1 * arr2  # Element-wise multiplication
arr1 / arr2
arr1 ** 2  # Squaring
arr1 % 3  # Modulo operation
```

Universal Functions (ufuncs)

Apply mathematical functions element-wise.

Aggregation Functions

Compute summary statistics across array dimensions.

```
# Basic statistics
np.sum(arr)
np.mean(arr)
np.std(arr)  # Standard deviation
np.var(arr)  # Variance
np.min(arr)
np.max(arr)
# Along specific axis
np.sum(arr2d, axis=0)  # Sum along rows
np.mean(arr2d, axis=1)  # Mean along columns
```

```
# Trigonometric functions
np.sin(arr)
np.cos(arr)
np.tan(arr)
# Exponential and logarithmic
np.exp(arr)
np.log(arr)
np.log10(arr)
# Square root and power
np.sqrt(arr)
np.power(arr, 3)
```

Comparison Operations

Element-wise comparisons returning boolean arrays.

```
# Comparison operators
arr > 5
arr == 3
arr != 0
# Array comparisons
np.array_equal(arr1, arr2)
np.allclose(arr1, arr2)  # Within tolerance
# Any/all operations
np.any(arr > 5)
np.all(arr > 0)
```

Linear Algebra

Matrix operations and linear algebra computations.

Matrix Operations: `np.dot()` / `@`

Perform matrix multiplication and dot products.

```
# Matrix multiplication
np.dot(A, B)
A @ B  # Python 3.5+ operator
# Element-wise multiplication
A * B
# Matrix power
np.linalg.matrix_power(A, 3)
```

Decompositions: `np.linalg`

Matrix decompositions for advanced computations.

```
# Eigenvalues and eigenvectors
eigenvals, eigenvecs = np.linalg.eig(A)
# Singular Value Decomposition
U, s, Vt = np.linalg.svd(A)
# QR decomposition
Q, R = np.linalg.qr(A)
```

Matrix Properties

Compute important matrix characteristics.

```
# Determinant
np.linalg.det(A)
# Matrix inverse
np.linalg.inv(A)
# Pseudo-inverse
np.linalg.pinv(A)
# Matrix rank
np.linalg.matrix_rank(A)
# Trace (sum of diagonal)
np.trace(A)
```

Solving Linear Systems: `np.linalg.solve()`

Solve systems of linear equations.

```
# Solve Ax = b
x = np.linalg.solve(A, b)
# Least squares solution
x = np.linalg.lstsq(A, b, rcond=None)[0]
```

Array Input/Output

Save and load arrays using various formats.

NumPy Binary: `np.save()` / `np.load()`

Efficient binary format for NumPy arrays.

```
# Save single array
np.save('array.npy', arr)
# Load array
loaded_arr = np.load('array.npy')
# Save multiple arrays
np.savez('arrays.npz', a=arr1, b=arr2)
# Load multiple arrays
data = np.load('arrays.npz')
arr1_loaded = data['a']
```

Text Files: `np.loadtxt()` / `np.savetxt()`

Read and write arrays as text files.

```
# Load from CSV/text file
arr = np.loadtxt('data.csv', delimiter=',')
# Skip header row
arr = np.loadtxt('data.csv', delimiter=',', skiprows=1)
# Save to text file
np.savetxt('output.csv', arr, delimiter=',', fmt='%.2f')
```

CSV with Structured Data: `np.genfromtxt()`

Advanced text file reading with missing data handling.

```
# Handle missing values
arr = np.genfromtxt('data.csv', delimiter=',',
                    missing_values='N/A', filling_values=0)
# Named columns
data = np.genfromtxt('data.csv', delimiter=',',
                     names=True, dtype=None)
```

Memory Mapping: `np.memmap()`

Work with arrays too large to fit in memory.

```
# Create memory-mapped array
mmap_arr = np.memmap('large_array.dat',
dtype='float32',
                     mode='w+', shape=(1000000,))
# Access like regular array but stored on disk
mmap_arr[0:10] = np.random.random(10)
```

Performance & Broadcasting

Optimize array operations and understand broadcasting rules.

Broadcasting Rules

Understand how NumPy handles operations on different shaped arrays.

```
# Broadcasting examples
arr1 = np.array([[1, 2, 3]])  # Shape (1, 3)
arr2 = np.array([[1], [2]])   # Shape (2, 1)
result = arr1 + arr2          # Shape (2, 3)

# Scalar broadcasting
arr + 5  # Adds 5 to all elements
arr * 2  # Multiplies all elements by 2
```

Vectorized Operations

Use NumPy's built-in functions instead of Python loops.

```
# Instead of loops, use vectorized operations
# Bad: for loop
result = []
for x in arr:
    result.append(x ** 2)

# Good: vectorized
result = arr ** 2

# Custom vectorized function
def custom_func(x):
    return x ** 2 + 2 * x + 1
vec_func = np.vectorize(custom_func)
result = vec_func(arr)
```

Memory Optimization

Techniques for efficient memory usage with large arrays.

```
# Use appropriate data types
arr_int8 = arr.astype(np.int8)  # 1 byte per element
arr_float32 = arr.astype(np.float32)  # 4 bytes vs 8 for
float64

# Views vs copies
view = arr[::2]  # Creates view (shares memory)
copy = arr[::2].copy()  # Creates copy (new memory)

# Check if array is view or copy
view.base is arr  # True for view
```

Performance Tips

Best practices for fast NumPy code.

```
# Use in-place operations when possible
arr += 5  # Instead of arr = arr + 5
np.add(arr, 5, out=arr)  # Explicit in-place

# Minimize array creation
# Bad: creates intermediate arrays
result = ((arr + 1) * 2) ** 2
# Better: use compound operations where possible
```

Random Number Generation

Generate random numbers and samples for simulations and testing.

Basic Random: `np.random`

Generate random numbers from various distributions.

```
# Random floats [0, 1)
np.random.random(5)
# Random integers
np.random.randint(0, 10,
size=5)
# Normal distribution
np.random.normal(mu=0,
sigma=1, size=5)
# Uniform distribution
np.random.uniform(-1, 1,
size=5)
```

Sampling: `choice()` / `shuffle()`

Sample from existing data or permute arrays.

```
# Random choice from array
np.random.choice(arr, size=3)
# Without replacement
np.random.choice(arr, size=3,
replace=False)
# Shuffle array in-place
np.random.shuffle(arr)
# Random permutation
np.random.permutation(arr)
```

Seeds & Generators

Control randomness for reproducible results.

```
# Set seed for reproducibility
np.random.seed(42)
# Modern approach: Generator
rng =
np.random.default_rng(42)
rng.random(5)
rng.integers(0, 10, size=5)
rng.normal(0, 1, size=5)
```

Statistical Functions

Comprehensive statistical operations on arrays.

Descriptive Statistics

Basic statistical measures of central tendency and spread.

```
# Central tendency
np.mean(arr)
np.median(arr)
# Spread measures
np.std(arr)  # Standard deviation
np.var(arr)  # Variance
np.ptp(arr)  # Peak to peak (max - min)
# Percentiles
np.percentile(arr, [25, 50, 75])
np.quantile(arr, [0.25, 0.5, 0.75])
```

Correlation & Covariance

Measure relationships between variables.

```
# Correlation coefficient
np.corrcoef(x, y)
# Covariance
np.cov(x, y)
```

Histogram & Binning

Analyze data distribution and create bins.

```
# Histogram
counts, bins = np.histogram(arr, bins=10)
# 2D histogram
H, xedges, yedges = np.histogram2d(x, y, bins=10)
# Digitize (assign bin indices)
bin_indices = np.digitize(arr, bins)
```

Special Statistical Functions

Advanced statistical computations.

```
# Weighted statistics
np.average(arr, weights=weights)
# Unique values and counts
unique_vals, counts = np.unique(arr,
return_counts=True)
# Bincount (for integer arrays)
np.bincount(int_arr)
```

```
# Cross-correlation
np.correlate(x, y, mode='full')
```

NumPy Installation & Setup

Install and configure NumPy for your Python environment.

Pip: `pip install numpy`

Standard Python package installer.

```
# Install NumPy
pip install numpy
# Upgrade to latest version
pip install numpy --upgrade
# Install specific version
pip install numpy==1.21.0
# Show package information
pip show numpy
```

Conda: `conda install numpy`

Package manager for Anaconda/Miniconda environments.

```
# Install NumPy in current environment
conda install numpy
# Update NumPy
conda update numpy
# Install from conda-forge
conda install -c conda-forge numpy
# Create environment with NumPy
conda create -n myenv numpy
```

Check Installation & Import

Verify your NumPy installation and standard import.

```
# Standard import
import numpy as np
# Check version
print(np.__version__)
# Check build information
np.show_config()
# Set print options
np.set_printoptions(precision=2, suppress=True)
```

Advanced Features

Specialized NumPy functionality for advanced use cases.

Structured Arrays

Arrays with named fields for complex data structures.

```
# Define structured data type
dt = np.dtype([('name', 'U10'), ('age', 'i4'), ('weight', 'f4')])
# Create structured array
people = np.array([('Alice', 25, 55.0), ('Bob', 30, 70.5)], dtype=dt)
# Access fields
people['name']
people['age']
```

Masked Arrays: `np.ma`

Handle arrays with missing or invalid data.

```
# Create masked array
masked_arr = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 0, 0])
# Operations ignore masked values
np.ma.mean(masked_arr)
# Fill masked values
filled = masked_arr.filled(0)
```

Polynomials: `np.poly1d`

Work with polynomial expressions and operations.

```
# Create polynomial (coefficients in descending order)
p = np.poly1d([1, -2, 1])  # x² - 2x + 1
# Evaluate polynomial
p(5)  # Evaluate at x=5
# Find roots
np.roots([1, -2, 1])
# Polynomial fitting
coeff = np.polyfit(x, y, degree=2)
```

Fast Fourier Transform: `np.fft`

Frequency domain analysis and signal processing.

```
# 1D FFT
fft_result = np.fft.fft(signal)
# Frequencies
freqs = np.fft.fftfreq(len(signal))
# Inverse FFT
reconstructed = np.fft.ifft(fft_result)
# 2D FFT for images
fft2d = np.fft.fft2(image)
```

Reference: This cheatsheet covers essential NumPy commands and modern practices for efficient numerical computing and array manipulation in data science workflows.