Complete NumPy Tutorial

Master the fundamental package for scientific computing in Python

Introduction to NumPy

NumPy (Numerical Python) is the fundamental package for scientific computing in Python. It provides a powerful N-dimensional array object, broadcasting functions, and tools for integrating with C/C++ and Fortran code.

Why NumPy?

NumPy arrays are faster and more memory-efficient than Python lists for numerical operations. They enable vectorized operations that eliminate the need for explicit loops in most cases.

Reshaping Arrays

Reshape and Flatten Operations

```
arr = np.arange(12)
print(arr)  # [0  1  2  3  4  5  6  7  8  9 10 11]

# Reshape to 2D
reshaped = arr.reshape(3, 4)
print(reshaped)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Reshape to 3D
reshaped_3d = arr.reshape(2, 2, 3)
print(reshaped_3d.shape)  # (2, 2, 3)

# Flatten back to 1D
flattened = reshaped.flatten()
print(flattened)  # [0  1  2  3  4  5  6  7  8  9 10 11]
```

Joining Arrays

Concatenate and Stack

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

# Concatenate
concat = np.concatenate([a, b])
print(concat)  # [1 2 3 4 5 6]

# Stack
vertical = np.vstack([a, b])    # Stack vertically
horizontal = np.hstack([a, b])  # Stack horizontally

print(vertical)
# [[1 2 3]
#  [4 5 6]]

print(horizontal)  # [1 2 3 4 5 6]
```

Broadcasting

Broadcasting allows NumPy to perform operations on arrays with different shapes without explicitly reshaping them.

Broadcasting Examples

```
# Scalar with array
arr = np.array([[1, 2, 3], [4, 5, 6]])
result = arr + 10
print(result)
# [[11 12 13]
#  [14 15 16]]

# Different shaped arrays
a = np.array([[1], [2], [3]])  # (3, 1)
b = np.array([10, 20, 30])     # (3,)
result = a + b
print(result)
# [[11 21 31]
#  [12 22 32]
#  [13 23 33]]
```

Broadcasting Rules

1. Arrays are aligned from the rightmost dimension
2. Dimensions with size 1 can be stretched
3. Missing dimensions are assumed to have size 1

Linear Algebra

Matrix Operations

```
# Matrix multiplication
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Dot product
dot_product = np.dot(A, B)
# or
dot_product = A @ B
print(dot_product)
# [[19 22]
#  [43 50]]

# Matrix operations
print(np.transpose(A))      # Transpose
print(np.linalg.det(A))     # Determinant: -2.0
print(np.linalg.inv(A))     # Inverse
print(np.trace(A))          # Trace: 5

# Eigenvalues and eigenvectors
eigenvals, eigenvecs = np.linalg.eig(A)
print(eigenvals)
```

Statistics and Aggregations

Statistical Functions

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

# Basic statistics
print(np.mean(arr))    # 3.5
print(np.median(arr))  # 3.5
print(np.std(arr))     # 1.71
print(np.var(arr))     # 2.92

# Aggregations
print(np.sum(arr))     # 21
print(np.min(arr))     # 1
print(np.max(arr))     # 6

# Along specific axes
print(np.sum(arr, axis=0))  # [5 7 9] (column sums)
print(np.sum(arr, axis=1))  # [6 15] (row sums)

# Other useful functions
print(np.argmax(arr))       # 5 (index of maximum)
print(np.argmin(arr))       # 0 (index of minimum)
print(np.unique(arr))       # [1 2 3 4 5 6]
```

Working with Different Data Types

Data Type Operations

```
# Specify data type during creation
int_arr = np.array([1, 2, 3], dtype=np.int32)
float_arr = np.array([1, 2, 3], dtype=np.float64)
bool_arr = np.array([True, False, True], dtype=np.bool_)

# Convert data types
arr = np.array([1.7, 2.3, 3.9])
int_converted = arr.astype(np.int32)
print(int_converted)  # [1 2 3]

# Check data type
print(arr.dtype)  # float64
```

File Input/Output

Saving and Loading Arrays

```
# Save array to file
arr = np.array([1, 2, 3, 4, 5])
np.save('my_array.npy', arr)

# Load array from file
loaded_arr = np.load('my_array.npy')

# Save/load text files
np.savetxt('data.txt', arr)
loaded_txt = np.loadtxt('data.txt')

# Save multiple arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
np.savez('multiple_arrays.npz', first=arr1, second=arr2)

# Load multiple arrays
loaded = np.load('multiple_arrays.npz')
print(loaded['first'])
print(loaded['second'])
```

Performance Tips

Vectorization vs Loops

Performance Comparison

```
# Slow: Using Python loops
def slow_sum(arr):
    total = 0
    for i in range(len(arr)):
        total += arr[i]
    return total

# Fast: Using NumPy vectorization
def fast_sum(arr):
    return np.sum(arr)

# Example timing (conceptual)
large_arr = np.random.random(1000000)
# fast_sum will be significantly faster
```

Best Practices

- Use vectorized operations instead of Python loops
- Avoid unnecessary copies - use views when possible
- Choose appropriate data types
- Use in-place operations when possible (+=, *=, etc.)
- Pre-allocate arrays when size is known

Memory Efficiency

Views vs Copies

```
# Use views instead of copies when possible
arr = np.arange(1000000)
view = arr[::2]  # This is a view, not a copy
copy = arr[::2].copy()  # This creates a copy

# Check if it's a view or copy
print(view.base is arr)  # True (it's a view)
print(copy.base is arr)  # False (it's a copy)
```

Real-World Examples

Image Processing Example

Simple Image Operations

```
# Simulate a grayscale image
image = np.random.randint(0, 256, (100, 100), dtype=np.uint8)

# Apply a simple filter (brighten)
brightened = np.clip(image + 50, 0, 255)

# Create a binary mask
mask = image > 128

print(f"Original image shape: {image.shape}")
print(f"Brightened pixels: {np.sum(brightened > image)}")
print(f"Pixels above threshold: {np.sum(mask)}")
```

Data Analysis Example

Statistical Analysis

```
# Generate sample data
data = np.random.normal(100, 15, 1000)  # Mean=100, std=15

# Basic analysis
print(f"Mean: {np.mean(data):.2f}")
print(f"Std: {np.std(data):.2f}")
print(f"95th percentile: {np.percentile(data, 95):.2f}")

# Find outliers (2 standard deviations from mean)
mean = np.mean(data)
std = np.std(data)
outliers = data[(data < mean - 2*std) | (data > mean + 2*std)]
print(f"Number of outliers: {len(outliers)}")
print(f"Outlier percentage: {len(outliers)/len(data)*100:.1f}%")
```

Financial Data Example

Stock Price Analysis

```
# Simulate stock prices
days = 100
initial_price = 100
daily_returns = np.random.normal(0.001, 0.02, days)  # 0.1% mean, 2% std
prices = initial_price * np.cumprod(1 + daily_returns)

# Calculate statistics
print(f"Starting price: ${initial_price:.2f}")
print(f"Ending price: ${prices[-1]:.2f}")
print(f"Total return: {(prices[-1]/initial_price - 1)*100:.2f}%")
print(f"Max price: ${np.max(prices):.2f}")
print(f"Min price: ${np.min(prices):.2f}")
print(f"Volatility (std of returns): {np.std(daily_returns)*100:.2f}%")
```

Conclusion

NumPy is the foundation of the scientific Python ecosystem. With this comprehensive tutorial, you've learned the essential concepts and operations that will serve as building blocks for more advanced data science and scientific computing tasks.

Key Takeaways

- **Arrays are faster and more memory-efficient** than Python lists for numerical operations
- **Vectorization eliminates the need for explicit loops** in most mathematical operations
- **Broadcasting allows operations between different shaped arrays** automatically
- **Rich functionality for mathematical operations** including linear algebra and statistics
- **Foundation for other scientific Python libraries** like Pandas, Matplotlib, and Scikit-learn

Next Steps

After mastering NumPy, consider exploring these complementary libraries:

- **Pandas** - For data manipulation and analysis with DataFrames
- **Matplotlib** - For creating static, animated, and interactive visualizations
- **SciPy** - For advanced scientific computing and optimization
- **Scikit-learn** - For machine learning algorithms and tools
- **Jupyter Notebooks** - For interactive data analysis and experimentation

Practice Makes Perfect

The best way to master NumPy is through hands-on practice. Try implementing the examples with your own data, experiment with different operations, and gradually work on more complex projects that combine multiple NumPy concepts.

Happy coding with NumPy!

>Key Features

- Powerful N-dimensional array object (ndarray)
- Broadcasting functions for different shaped arrays
- Linear algebra, Fourier transform, and random number capabilities

- Tools for integrating with C/C++ and Fortran code

Installation

Installation Commands

```
# Using pip
pip install numpy

# Using conda
conda install numpy
```

Import NumPy

```
import numpy as np
```

NumPy Arrays Basics

Creating Your First Array

Basic Array Creation

```
import numpy as np

# From a Python list
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))

# 2D array from nested lists
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d)
```

Output:

```
[1 2 3 4 5] <class 'numpy.ndarray'> [[1 2 3] [4 5 6]]
```

Key Differences from Python Lists

Lists vs Arrays

```
# Python list multiplication
python_list = [1, 2, 3]
print(python_list * 2)  # [1, 2, 3, 1, 2, 3]

# NumPy array multiplication
numpy_array = np.array([1, 2, 3])
print(numpy_array * 2)  # [2 4 6]
```

Important Note

NumPy arrays perform element-wise operations, while Python lists concatenate or repeat elements. This is a fundamental difference that makes NumPy powerful for mathematical operations.

Array Creation

Common Creation Functions

Array Creation Methods

```
# Zeros and ones
zeros = np.zeros(5)           # [0. 0. 0. 0. 0.]
ones = np.ones((2, 3))        # 2x3 array of ones
full = np.full((2, 2), 7)     # 2x2 array filled with 7

# Range functions
arange = np.arange(0, 10, 2)    # [0 2 4 6 8]
linspace = np.linspace(0, 1, 5) # [0.   0.25 0.5  0.75 1.  ]

# Identity matrix
identity = np.eye(3)          # 3x3 identity matrix

# Random arrays
random_arr = np.random.random((2, 3))  # Random values between 0 and 1
random_int = np.random.randint(0, 10, (2, 3))  # Random integers
```

Pro Tip

Use `np.linspace()` when you need a specific number of evenly spaced points, and `np.arange()` when you need a specific step size.

Array Attributes

Exploring Array Properties

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(f"Shape: {arr.shape}")        # (2, 4)
print(f"Size: {arr.size}")          # 8
print(f"Dimensions: {arr.ndim}")    # 2
print(f"Data type: {arr.dtype}")    # int64 (may vary by system)
print(f"Item size: {arr.itemsize}") # 8 bytes
print(f"Total bytes: {arr.nbytes}") # 64 bytes
```

Output:

```
Shape: (2, 4) Size: 8 Dimensions: 2 Data type: int64 Item size: 8 Total bytes: 64
```

Array Indexing and Slicing

Basic Indexing

1D Array Indexing

```
arr = np.array([0, 1, 2, 3, 4, 5])

# Single element
print(arr[0])    # 0
print(arr[-1])   # 5

# Slicing
print(arr[1:4])  # [1 2 3]
print(arr[::2])  # [0 2 4] (every second element)
```

2D Array Indexing

2D Array Operations

```
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Element access
print(arr_2d[0, 1])    # 2
print(arr_2d[1, :])    # [4 5 6] (entire row)
print(arr_2d[:, 1])    # [2 5 8] (entire column)

# Subarray
print(arr_2d[0:2, 1:3])
# [[2 3]
#  [5 6]]
```

Boolean Indexing

Boolean Masks and Filtering

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

# Boolean mask
mask = arr > 3
print(mask)        # [False False False  True  True]
print(arr[mask])   # [4 5]

# Direct boolean indexing
print(arr[arr > 3])  # [4 5]
print(arr[(arr > 2) & (arr < 5)])  # [3 4]
```

Array Operations

Element-wise Operations

Arithmetic Operations

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

# Arithmetic operations
print(a + b)   # [6  8 10 12]
print(a - b)   # [-4 -4 -4 -4]
print(a * b)   # [5 12 21 32]
print(a / b)   # [0.2 0.33 0.43 0.5]
print(a ** 2)  # [1  4  9 16]

# Comparison operations
print(a > 2)   # [False False  True  True]
print(a == b)  # [False False False False]
```

Universal Functions (ufuncs)

Mathematical Functions

```
arr = np.array([1, 4, 9, 16])

print(np.sqrt(arr))    # [1. 2. 3. 4.]
print(np.log(arr))     # [0.   1.39 2.20 2.77]
print(np.exp(arr))     # [2.72e+00 5.46e+01 8.10e+03 8.89e+06]
print(np.sin(arr))     # [0.84  -0.76  0.41  -0.29]
```

Mathematical Functions

Trigonometric Functions

Trigonometry with NumPy

```
angles = np.array([0, np.pi/4, np.pi/2, np.pi])

print(np.sin(angles))  # [0.    0.71  1.    0.  ]
print(np.cos(angles))  # [1.    0.71  0.   -1.  ]
print(np.tan(angles))  # [0.    1.    inf   0.  ]
```

Rounding Functions

Number Rounding Operations

```
arr = np.array([1.2, 2.7, 3.1, 4.9])

print(np.round(arr))    # [1. 3. 3. 5.]
print(np.floor(arr))    # [1. 2. 3. 4.]
print(np.ceil(arr))     # [2. 3. 4. 5.]
print(np.trunc(arr))    # [1. 2. 3. 4.]
```

Array Manipulation