

NumPy Fundamentals: The Foundation of Python Data Science

Welcome to this comprehensive guide on NumPy, the fundamental package for scientific computing in Python. This presentation will walk you through the essential concepts and techniques that make NumPy the backbone of data analysis, machine learning, and scientific computing in the Python ecosystem.

Creating NumPy Arrays

NumPy arrays are the foundation of the library, offering powerful ways to store and manipulate data efficiently.

1

np.array()

Creates an array from a Python list or tuple

```
import numpy as np
arr = np.array([1, 2, 3, 4]) #
1D array
arr2 = np.array([[1, 2], [3,
4]]) # 2D array
```

2

np.zeros() & np.ones()

Creates arrays filled with zeros or ones

```
zeros = np.zeros((3, 4)) # 3x4
array of zeros
ones = np.ones(5) # 1D array
of five ones
```

3

np.arange() & np.linspace()

Creates sequences of numbers

```
seq1 = np.arange(0, 10, 2) #
[0, 2, 4, 6, 8]
seq2 = np.linspace(0, 1, 5) #
5 evenly spaced points
```

These array creation functions provide the building blocks for more complex data structures and computations in NumPy.

Array Indexing: Accessing Individual Elements

NumPy arrays use zero-based indexing, just like Python lists. This means the first element is at index 0, the second at index 1, and so on.

Negative indices count from the end of the array, with -1 referring to the last element.

Pro Tip: NumPy indexing is much faster than Python list indexing because NumPy is implemented in C.

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
# Accessing elements
first = arr[0] # 10
last = arr[-1] # 50
third = arr[2] # 30
```



Array Slicing: Selecting Ranges of Elements

Slicing lets you extract subarrays by specifying a range of indices using the notation `start:stop:step`.



Basic Syntax

`array[start:stop:step]`

- **start**: First index (inclusive, default 0)
- **stop**: End index (exclusive, default array length)
- **step**: Increment between indices (default 1)



Examples

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# First five elements
arr[:5]    # [0, 1, 2, 3, 4]
# Elements from index 5 to the end
arr[5:]    # [5, 6, 7, 8, 9]
# Every second element
arr[::2]   # [0, 2, 4, 6, 8]
```

Slicing creates views, not copies, which means modifying a slice will modify the original array. Use `arr.copy()` if you need an independent copy.

Accessing Subarrays in Multidimensional Arrays

For multidimensional arrays, you can use comma-separated indices or slices to access specific elements or subarrays.

```
# Create a 3x4 array
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])

# Access a single element (row 0, column 2)
element = arr[0, 2]    # 3

# Access a subarray (rows 0-1, columns 1-2)
subarray = arr[0:2, 1:3]
# array([[2, 3],
#        [6, 7]])

# All rows, every second column
subarray2 = arr[:, ::2]
# array([[ 1,  3],
#        [ 5,  7],
#        [ 9, 11]])
```

Understanding multidimensional indexing is crucial for efficiently working with matrices, tensors, and other higher-dimensional data structures.



Reshaping Arrays: Changing Dimensions

The `reshape()` method allows you to change the shape of an array without changing its data. The new shape must have the same total number of elements as the original array.

Common reshaping operations include:

- Converting between 1D and 2D arrays
- Restructuring data for analysis or visualization
- Preparing data for machine learning algorithms

Use `-1` as a dimension size to let NumPy automatically calculate the appropriate value.

Original Array

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

Reshape to 3×4

```
arr.reshape(3, 4)
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
```

Reshape to 2×2×3

```
arr.reshape(2, 2, 3)
# array([[[ 0,  1,  2],
#         [ 3,  4,  5]],
#        [[ 6,  7,  8],
#         [ 9, 10, 11]]])
```

Array Concatenation and Splitting

NumPy provides functions to join arrays together and split them into multiple subarrays.

Concatenation Functions

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

# Concatenate along rows (axis=0)
np.concatenate((a, b), axis=0)
# array([[1, 2],
#        [3, 4],
#        [5, 6],
#        [7, 8]])

# Concatenate along columns (axis=1)
np.concatenate((a, b), axis=1)
# array([[1, 2, 5, 6],
#        [3, 4, 7, 8]])

# Alternatives:
# np.vstack((a, b)) # Vertical stack
# np.hstack((a, b)) # Horizontal stack
```

Splitting Functions

```
# Create a sample array
arr = np.arange(16).reshape(4, 4)

# Split into 2 equal arrays horizontally
np.split(arr, 2, axis=0)
# [array([[0, 1, 2, 3],
#        [4, 5, 6, 7]]),
#   array([[ 8,  9, 10, 11],
#        [12, 13, 14, 15]])]

# Split into arrays at specific indices
np.split(arr, [1, 3], axis=1)
# [array([[ 0],
#        [ 4],
#        [ 8],
#        [12]]),
#   array([[ 1,  2],
#        [ 5,  6],
#        [ 9, 10],
#        [13, 14]]),
#   array([[ 3],
#        [ 7],
#        [11],
#        [15]])]
```

These operations are essential for data preprocessing, feature engineering, and working with complex datasets.

Universal Functions (ufuncs): Vectorized Operations

NumPy's universal functions (ufuncs) perform element-wise operations on arrays, allowing for vectorized computations that are much faster than Python loops.



Mathematical Functions

NumPy provides vectorized versions of mathematical operations:

```
arr = np.array([1, 4, 9, 16, 25])
np.sqrt(arr)  # [1, 2, 3, 4, 5]
np.exp(arr)   # Exponential e^x
np.log(arr)   # Natural logarithm
np.sin(arr)   # Sine function
```



Comparison Operations

Compare entire arrays element-wise:

```
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])
a == b  # [False, True, False, True]
a > b   # [False, False, True, False]
np.maximum(a, b)  # [4, 2, 3, 4]
```



Bitwise Operations

Perform bitwise operations on integer arrays:

```
x = np.array([1, 2, 3, 4])
y = np.array([4, 3, 2, 1])
np.bitwise_and(x, y)  # Bitwise AND
np.bitwise_or(x, y)   # Bitwise OR
```

Ufuncs are highly optimized in C and can utilize parallel processing capabilities for significant performance gains over traditional Python operations.

The Power of Vectorization: Avoiding Loops

One of NumPy's greatest strengths is vectorization — performing operations on entire arrays without explicit loops. This approach is significantly faster due to optimized C implementations and potential parallelization.

Python Loop (Slow)

```
import time
import numpy as np

# Create large arrays
size = 1000000
data = np.random.random(size)
result_l = np.zeros(size)
result_v = np.zeros(size)

# Time a Python loop
start = time.time()
for i in range(size):
    result_l[i] = data[i] ** 2
loop_time = time.time() - start
```


NumPy Vectorization (Fast)

```
# Time vectorized operation
start = time.time()
result_v = data ** 2
vector_time = time.time() - start

# Compare times
print(f"Loop time: {loop_time:.6f} seconds")
print(f"Vector time: {vector_time:.6f} seconds")

print(result_l)
print(result_v)

print(f"Speedup: {loop_time/vector_time:.1f}x")
# Typical output:
# Loop time: 0.584321 seconds
# Vector time: 0.003124 seconds
# Speedup: 187.0x
```

 **Common Pitfall:** Using Python loops with NumPy arrays can negate most of NumPy's performance benefits. Always look for vectorized alternatives to loops.

Aggregation and Statistical Functions

NumPy provides powerful functions for summarizing and analyzing data across entire arrays or along specific axes.

np.sum()

Summation

Calculate the sum of all elements or along a specific axis.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
np.sum(arr)          # 21 (all elements)
np.sum(arr, axis=0) # [5, 7, 9] (column
sums)
np.sum(arr, axis=1) # [6, 15] (row sums)
```

np.min()

Minimum Values

Find the minimum value in an array.

```
np.min(arr) # 1 (overall minimum)
np.min(arr, axis=0) # [1, 2, 3] (column
mins)
arr.min(axis=1)    # [1, 4] (row mins)
```

np.max()

Maximum Values

Find the maximum value in an array.

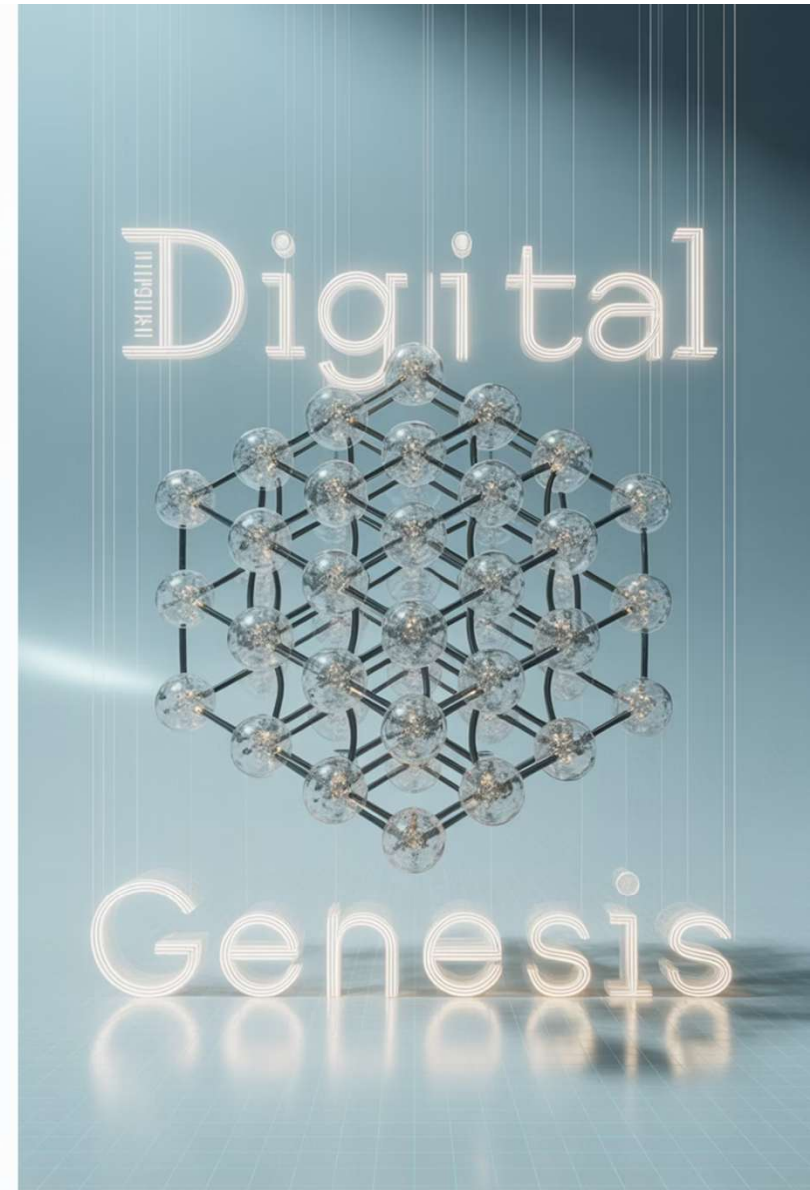
```
np.max(arr) # 6 (overall maximum)
np.max(arr, axis=0) # [4, 5, 6] (column
maxs)
arr.max(axis=1) # [3, 6] (row maxs)
```

Additional functions include `np.mean()`, `np.median()`, `np.std()` (standard deviation), and `np.var()` (variance). These functions form the foundation of statistical analysis in NumPy and are essential for data science workflows.

Remember that all these functions can operate along specific axes, making it easy to analyze data across rows, columns, or other dimensions in multidimensional arrays.

NumPy: Mastering Array Computation

Welcome to this guide on NumPy array operations and advanced indexing techniques. NumPy revolutionizes numerical computing in Python by enabling efficient array manipulation and lightning-fast mathematical operations. This presentation will cover vectorized operations, broadcasting, boolean indexing, and more to help you write faster, cleaner Python code for numerical analysis.



Computation on Arrays

Vectorized Operations

NumPy performs operations on entire arrays without explicit loops, making code more readable and drastically faster than traditional Python loops.

```
# Element-wise addition
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b # [5, 7, 9]
```

Universal Functions

NumPy's ufuncs operate element-wise on arrays, supporting various mathematical operations efficiently.

```
# Using ufuncs
import numpy as np
arr = np.array([0, np.pi/2, np.pi])
np.sin(arr) # [0., 1., 0.]
```

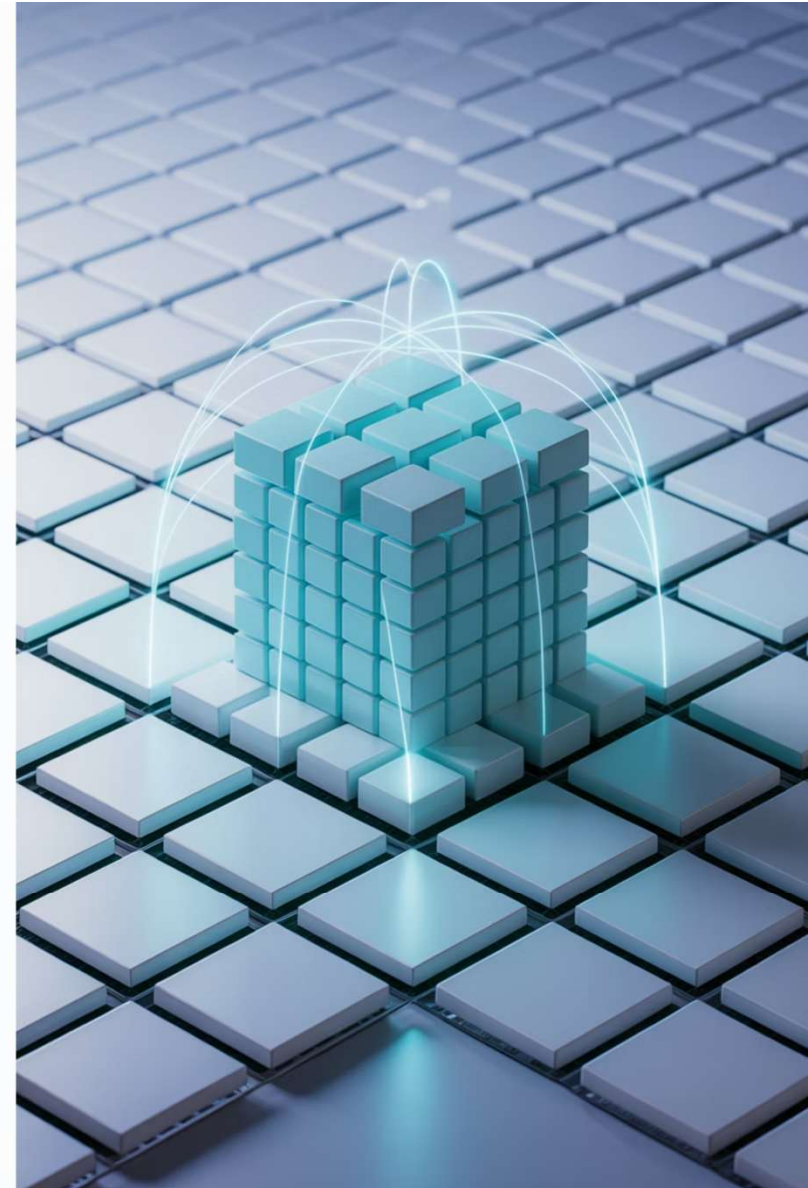
These vectorized operations leverage optimized C implementations behind the scenes, making NumPy 10-100x faster than equivalent Python code with loops. This performance gain is critical for data science and scientific computing applications.

Broadcasting: Operate on Arrays of Different Shapes

Broadcasting is NumPy's powerful mechanism that allows arithmetic operations between arrays of different shapes without creating unnecessary copies of data. This saves memory and computation time while making your code more concise.

```
# Broadcasting example
import numpy as np
a = np.array([1, 2, 3])      # Shape: (3,)
b = np.array([[1], [2], [3]]) # Shape: (3, 1)
c = a + b                    # Shape: (3, 3)
# Result:
# [[2 3 4]
#  [3 4 5]
#  [4 5 6]]
```

Broadcasting automatically "stretches" the smaller array to match the shape of the larger one, enabling element-wise operations without explicitly reshaping arrays or creating temporary copies.



Rules of Broadcasting

Rule 1: Pad with Ones

If array shapes differ in number of dimensions, pad the smaller shape with ones on the left until both shapes have the same length.

```
a.shape = (3,)    → (1, 3)
b.shape = (2, 1)  → (2, 1)
```

Rule 2: Compare Dimensions

Compare dimensions element-wise from right to left. Each dimension must either match exactly or one must be 1.

```
(1, 3) and (2, 1) → Compatible
(2, 3) and (2, 1) → Compatible
(3, 3) and (2, 1) → Compatible
```

Rule 3: Calculate Result Shape

The resulting shape is the maximum in each dimension.

```
(1, 3) and (2, 1) → (2, 3)
(2, 3) and (2, 1) → (2, 3)
(3, 3) and (2, 1) → (3, 3)
```

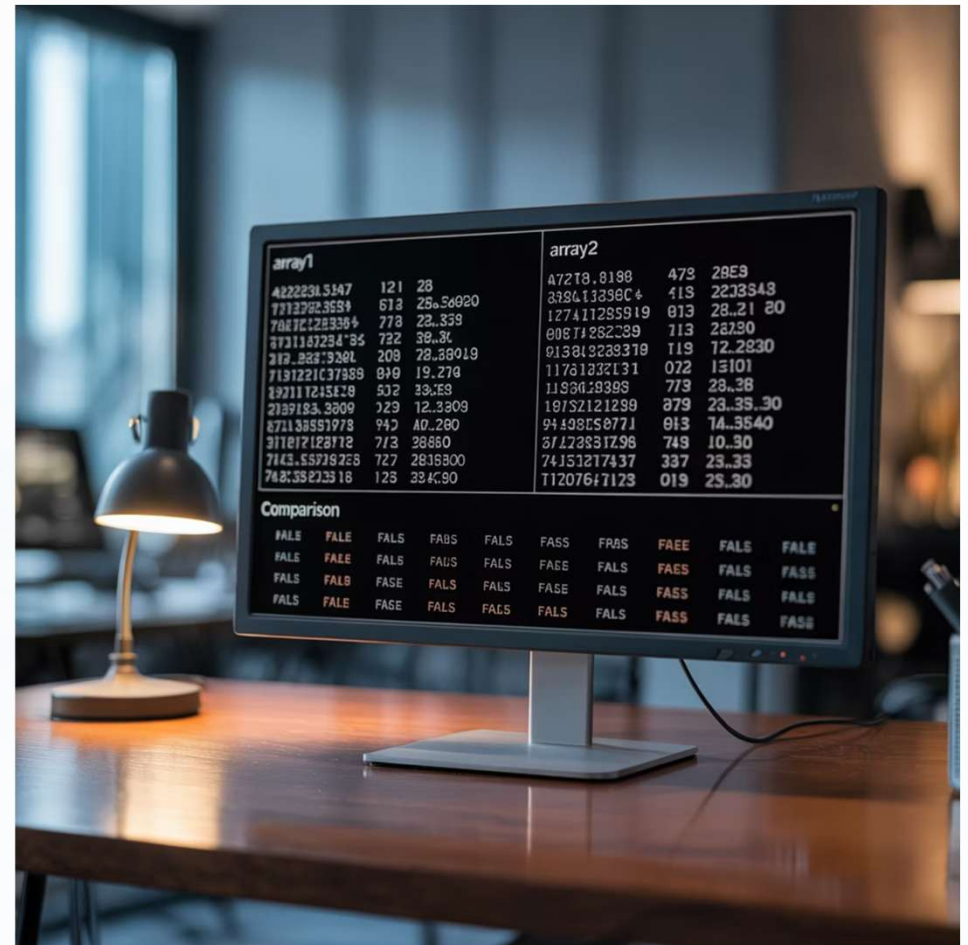
Comparisons and Boolean Arrays

Element-wise Comparisons

NumPy supports all standard comparison operators: <, >, <=, >=, ==, !=

```
import numpy as np  
arr = np.array([1, 2, 3, 4, 5])  
comparison = arr > 3  
comparison = [False, False, False, True, True]
```

These operations return boolean arrays of the same shape as the input.



Comparison operations create boolean arrays that can be used for filtering data or conditional

Masks and Boolean Logic

Creating Masks

Boolean arrays act as masks to filter data based on conditions.

```
data = np.array([1, 2, 3, 4, 5])
mask = data > 3 # [False, False, False, True, True]
```

Combining Masks

Use logical operators to combine multiple conditions:

- `&` for logical AND
- `|` for logical OR
- `~` for logical NOT

```
mask1 = (data > 2) & (data < 5) # AND
mask2 = (data < 2) | (data > 4) # OR
mask3 = ~(data == 3) # NOT
```

Important: Use `&`, `|`, and `~` instead of `and`, `or`, and `not` when working with NumPy boolean arrays!

Working with Boolean Arrays

Filtering with Boolean Indexing

```
import numpy as np
data = np.array([1, 2, 3, 4, 5])
# Extract values greater than 3
result = data[data > 3]
# [4, 5]
# More complex condition
result = data[(data > 2) & (data < 5)]
# [3, 4]
```

Boolean indexing selects only the elements where the corresponding mask value is `True`.

Conditional Modifications

```
# Replace all values > 3 with 10
data = np.array([1, 2, 3, 4, 5])
data[data > 3] = 10
# data is now [1, 2, 3, 10, 10]
# Conditional replacement
data = np.array([1, 2, 3, 4, 5])
data[data < 3] = 0
data[data >= 3] += 100
# data is now [0, 0, 103, 104, 105]
```

Fancy Indexing

Fancy indexing allows you to access multiple elements at once using arrays of indices, offering powerful data selection capabilities beyond what's possible with slicing.

```
import numpy as np
data = np.array([10, 20, 30, 40, 50, 60, 70, 80])
# Select specific elements by their indices
indices = np.array([0, 2, 5, 7])
selected = data[indices]  # [10, 30, 60, 80]
# Works with multidimensional arrays
toomatrix = np.arange(16).reshape(4, 4)
row_indices = np.array([0, 1, 3])
col_indices = np.array([1, 2, 0])
selected = toomatrix[row_indices, col_indices]  # [1, 6, 12]
```

Unlike slicing which returns a view, fancy indexing returns a [copy](#) of the selected data. This is important to remember when modifying elements.