Data Sprout

NUMPY FOR DATA SCIENCE

Welcome to this Numpy guide for data science! Numpy is a fundamental tool for numerical computing in Python and is essential for data science. This guide brings together key Numpy use cases that will help you work more efficiently with data.

The goal is to provide a simple, practical reference that will help you get familiar with the most important Numpy techniques used in data science. From basic array operations to more advanced functions, this guide is designed to give you the knowledge you need to confidently apply Numpy in your projects.

Explore the examples in this guide, **experiment with them in your own work**, and make these techniques part of your daily toolkit. Once you feel comfortable, **read more about the techniques** to expand your knowledge even further.

This guide is a great starting point, and I'm confident it will help you unlock new ways to work smarter with data. Enjoy the journey, and happy learning!

Basic Use Cases:

1. Array Creation
2. Basic Indexing and Slicing
3. Array Reshaping
4. Element-wise Operations
5. Broadcasting
6. Aggregations – `sum()`, `mean()`, `std()`, `min()`, `max()`
7. Boolean Indexing – Selecting elements based on conditions
8. Boolean Indexing with 2D Arrays
9. Sorting Arrays – `np.sort()`, `np.argsort()`
10. Finding Unique Elements
11. Random Sampling
12. Random Number Generation

Advanced NumPy Use Cases:

1. Stacking and Concatenating Arrays (Consider the Concatenation part in Basic)
2. Fancy Indexing
3. Vectorized Operations
4. Vectorized Custom Functions
5. Random Number Generation – Using `np.random` for generating random numbers, arrays
6. Statistical Functions – `np.percentile()`, `np.median()`, `np.corrcoef()`, `np.cov()`
7. Linear Algebra Operations – Matrix multiplication, determinants, inverses
8. Tile and Repeat – for repeating arrays
9. Meshgrid Functionality
10. Polynomial Fitting
11. Meshgrid Functionality – Using `np.meshgrid()` for constructing grids
12. Memory-Mapped Arrays
13. Array Transposition and Swapping Axes

Basic NumPy Use Cases

---

1. Array Creation

Creating arrays is fundamental in NumPy. We can create arrays with various methods like `np.array()`, `np.zeros()`, `np.ones()`, `np.arange()`, and `np.linspace()`.

Code:

```python
import numpy as np

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

# Zeroes and ones arrays
zeros = np.zeros((3, 3))
ones = np.ones((2, 4))

# Ranges of values
arange_array = np.arange(0, 10, 2)
linspace_array = np.linspace(0, 1, 5)
print(arr, zeros, ones, arange_array, linspace_array, sep='\n\n')
```

Output:

```lua
[1 2 3 4 5]

[[0. 0. 0.]
[0. 0. 0.]
[0. 0. 0.]]

[[1. 1. 1. 1.]
[1. 1. 1. 1.]]

[0 2 4 6 8]

[0.   0.25  0.5   0.75  1.  ]
```

2. Basic Indexing and Slicing

Indexing and slicing help you access specific parts of an array, which is very useful for filtering and transforming data.

Code:

```python
arr = np.array([10, 20, 30, 40, 50])

# Accessing elements
element = arr[2]    # Output: 30

# Slicing
slice_arr = arr[1:4]    # Output: [20 30 40] print(element,
slice_arr)
```

Output:

```csharp
30
[20 30 40]
```

3. Array Reshaping

Reshaping arrays allows you to change the dimensions of arrays without changing the data. This is useful when preparing data for models.

Code:

```python
arr = np.arange(1, 10)

reshaped_arr = arr.reshape(3, 3)
flattened_arr = reshaped_arr.ravel()   # or arr.flatten()

print(reshaped_arr)
print(flattened_arr)
```

Output:

```lua
[[1 2 3]
[4 5 6]
[7 8 9]]

[1 2 3 4 5 6 7 8 9]
```

4. Element-wise Operations

NumPy supports element-wise operations like addition, subtraction, and multiplication, which are critical when manipulating datasets.

Code:

```python
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise operations
added = arr1 + arr2  # Output: [5 7 9]
multiplied = arr1 * arr2  # Output: [4 10 18]
print(added, multiplied)
```

Output:

```css
[5 7 9]
[ 4 10 18]
```

5. Broadcasting

Broadcasting allows NumPy to work with arrays of different shapes during arithmetic operations, saving memory and computation time.

Code:

```python
arr1 = np.array([1, 2, 3])
arr2 = np.array([[1], [2], [3]])
broadcasted_sum = arr1 + arr2
print(broadcasted_sum)
```

Output:

```lua
[[2 3 4]
[3 4 5]
[4 5 6]]
```

6. Aggregations

Aggregation functions like `sum()`, `mean()`, `std()`, etc., help summarize data, which is useful in exploratory data analysis (EDA).

Code:

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

sum_val = arr.sum()    # Output: 15
mean_val = arr.mean()    # Output: 3.0
std_val = arr.std()    # Output: 1.4142
```

7. Boolean Indexing

Boolean indexing allows filtering based on conditions, which is often required for data preprocessing tasks like handling outliers.

Code:

```python
arr = np.array([1, 2, 3, 4, 5])
filtered_arr = arr[arr > 3]    # Output: [4 5]
print(filtered_arr)
```

Output:

```csharp
[4 5]
```

8. Boolean Indexing with 2D Arrays

Boolean indexing lets you filter elements from arrays based on conditions.

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

# Boolean indexing
bool_index = arr[arr > 5]
print("Boolean Indexing Result:", bool_index)
```

Output:

```plaintext
Boolean Indexing Result: [6 7 8 9]
```

9. Sorting Arrays

Sorting arrays is useful for ranking or organizing data before further analysis.

Code:

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

sorted_arr = np.sort(arr)  # Output: [1 2 3 4 5]
sorted_indices = np.argsort(arr)  # Output: [1 2 0 4 3]
print(sorted_arr, sorted_indices)
```

Output:

```csharp
[1 2 3 4 5]
[1 2 0 4 3]
```

---

10. Finding Unique Elements

`np.unique()` finds unique elements and their frequencies.

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

# Unique elements
unique_elements, counts = np.unique(arr, return_counts=True)
print("Unique Elements:", unique_elements)
print("Counts:", counts)
```

Output:

```plaintext
Unique Elements: [1 2 3]
Counts: [1 2 3]
```

---

11. Random Sampling

`np.random.choice()` allows random sampling with or without replacement.

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

# Random sampling
sample = np.random.choice(arr, size=3, replace=False)
print("Random Sample:", sample)
```

Output:

```plaintext
Random Sample: [2 4 5]
```

12. Random Number Generation

`np.random` can generate random numbers and arrays, useful for simulations and testing.

```python
# Random integers
rand_ints = np.random.randint(0, 10, size=(2, 3))

print("Random Integers:\n", rand_ints)

# Random floats between 0 and 1
rand_floats = np.random.rand(3, 3)
print("Random Floats:\n", rand_floats)
```

Output:

```plaintext
Random Integers:
 [[5 7 3]
 [2 8 0]]

Random Floats:
 [[0.523 0.846 0.129]
 [0.938 0.217 0.464]
 [0.731 0.197 0.852]]
```

Advanced NumPy Use Cases

---

1. Stacking and Concatenating Arrays

`np.hstack()`, `np.vstack()`, and `np.concatenate()` allow us to join multiple arrays together. While `hstack` stacks arrays horizontally, `vstack` stacks them vertically, and `concatenate` is more flexible, allowing concatenation along any axis.

```python
import numpy as np

# Arrays
arr1 = np.array([[1, [1]], [3, 4]])

arr2 = np.array([[5, 6]])

# Horizontal stacking
hstacked = np.hstack((arr1, arr2.T))

print("Horizontal Stacking:\n", hstacked)

# Vertical stacking
vstacked = np.vstack((arr1, arr2))

print("Vertical Stacking:\n", vstacked)

# Concatenation along axis 0
concatenated = np.concatenate((arr1, arr2), axis=0)
print("Concatenation:\n", concatenated)
```

Output:

```plaintext
Horizontal Stacking:
 [[1 2 5]
 [3 4 6]]

Vertical Stacking:
 [[1 2]
 [3 4]
 [5 6]]

Concatenation:
 [[1 2]
 [3 4]
 [5 6]]
```

2. Fancy Indexing with 2D Arrays

Fancy indexing allows access to specific elements in arrays by passing a list or array of indices.

```python
arr = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

# Fancy indexing
rows = [0, 1, 2]
cols = [1, 0, 2]
fancy_indexed = arr[rows, cols]
print("Fancy Indexing Result:", fancy_indexed)
```

Output:

```plaintext
Fancy Indexing Result: [20 40 90]
```

---

3. Vectorized Operations

Vectorized operations in NumPy optimize calculations by applying functions elementwise across arrays without explicit loops (for, while etc).

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

# Vectorized operations
squared = arr ** 2
print("Vectorized Squaring:", squared)
# Using np.add to perform addition


added = np.add(arr, 10)


print("Vectorized Addition:", added)
```

Output:

```plaintext
Vectorized Squaring: [ 1  4  9 16 25]
Vectorized Addition: [11 12 13 14 15]
```

---

4. Vectorized Custom Functions

You can vectorize custom functions for optimized array operations.

```python
# Custom function to cube elements
def custom_func(x):

    return x ** 3

# Vectorized version
vec_func = np.vectorize(custom_func)

arr = np.array([1, 2, 3, 4])
result = vec_func(arr)
print("Vectorized Custom Function:", result)
```

Output:

```plaintext
Vectorized Custom Function: [ 1  8 27 64]
```

---

5. Random Number Generation

`np.random` can generate random numbers and arrays, useful for simulations and testing.

```python
# Random integers
rand_ints = np.random.randint(0, 10, size=(2, 3))

print("Random Integers:\n", rand_ints)

# Random floats between 0 and 1
rand_floats = np.random.rand(3, 3)
print("Random Floats:\n", rand_floats)
```

Output:

```plaintext
Random Integers:
 [[5 7 3]
 [2 8 0]]

Random Floats:
 [[0.523 0.846 0.129]
 [0.938 0.217 0.464]
 [0.731 0.197 0.852]]
```

---

6. Statistical Functions

Functions like `np.percentile()`, `np.median()`, `np.corrcoef()`, and `np.cov()` allow statistical analysis on arrays.

```python
arr = np.array([10, 20, 30, 40, 50])

# Percentile
print("90th Percentile:", np.percentile(arr, 90))

# Median
print("Median:", np.median(arr))

# Correlation coefficient
data1 = np.array([1, 2, 3])
data2 = np.array([4, 5, 6])
print("Correlation Coefficient:\n", np.corrcoef(data1, data2))

# Covariance
print("Covariance Matrix:\n", np.cov(data1, data2))
```

Output:

```plaintext
90th Percentile: 46.0
Median: 30.0 Correlation
Coefficient:
 [[1. 1.]
 [1. 1.]]
Covariance Matrix:
 [[1. 1.]
 [1. 1.]]
```

---

7. Linear Algebra Operations

`np.linalg` provides functions for matrix multiplication, determinants, inverses, etc.

```python
matrix = np.array([[1, 2], [3, 4]])
# Matrix multiplication
print("Matrix Multiplication:\n",

np.dot(matrix, matrix))

# Determinant
print("Determinant:", np.linalg.det(matrix))

# Inverse print("Inverse:\n",



np.linalg.inv(matrix))
```

Output:

```plaintext
Matrix Multiplication:
 [[ 7 10]
 [15 22]]
Determinant: -2.0
Inverse:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

---

8. Tile and Repeat

`np.tile()` and `np.repeat()` repeat elements of arrays to create larger ones.

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

# Tiling
tiled = np.tile(arr, (2, 2))

print("Tiled Array:\n", tiled)

# Repeating
repeated = np.repeat(arr, 2)
print("Repeated Array:", repeated)
```

Output:

```plaintext
Tiled Array:
 [[1 2 3 1 2 3]
 [1 2 3 1 2 3]]
Repeated Array: [1 1 2 2 3 3]
```

---

9. Meshgrid Functionality

`np.meshgrid()` is used to create a grid of points for evaluating functions.

```python
x = np.array([1, 2, 3])
y = np.array([4, 5]) X,
Y = np.meshgrid(x, y)

print("X:\n", X)
print("Y:\n", Y)
```

Output:

```plaintext
X:
 [[1 2 3]
 [1 2 3]]
Y:
 [[4 4 4]
 [5 5 5]]
```

---

10. Polynomial Fitting

Using `np.polyfit()`, you can fit a polynomial to data for curve fitting and regression purposes. Polynomial fitting allows you to model non-linear relationships in data, and `np.polyfit()` finds the best-fitting polynomial of the specified degree.

Sample Code:

```python
import matplotlib.pyplot as plt

# Data points
x = np.array([0, 1, 2, 3, 4, 5]) y =
np.array([1, 1.8, 3.2, 4.1, 6.1, 7.9])

# Polynomial fitting of degree 2
coefficients = np.polyfit(x, y, 2)

poly = np.poly1d(coefficients)

# Plotting
xp = np.linspace(0, 5, 100)
plt.plot(x, y, 'o', label='Data points')
plt.plot(xp, poly(xp), '-', label='Fitted polynomial')
plt.legend() plt.show()
```

11. Memory-Mapped Arrays

`np.memmap()` allows you to work with large datasets that won't fit into memory by mapping a portion of a large file to an array. With `np.memmap()`, you can work with arrays larger than your system's RAM by storing them on disk and accessing portions as needed.

```python
# Create a memory-mapped array
large_array = np.memmap('large_data.dat', dtype='float32', mode='w+', shape=(1000000,))

# Writing data
large_array[:1000] = np.random.rand(1000)

# Accessing the data without loading it entirely into memory
subset = large_array[500:600]
print("Subset of memory-mapped array:", subset)
```

Output:

```sql
Subset of memory-mapped array: [0.87234564 0.43712312 0.21891234 ...]
```

12. Array Transposition and Swapping Axes

Transposing and swapping axes are essential for rearranging dimensions in NumPy arrays, which is crucial when handling multi-dimensional data like tensors.

`transpose()` rearranges the axes of the array.

`swapaxes()` swaps two specified axes.

`moveaxis()` moves one or more axes to a different position.

Sample Code:

```python
import numpy as np

# 3D Array
arr = np.random.rand(2, 3, 4)

# Transpose the array (swap axes)
transposed_arr = np.transpose(arr, (2, 1, 0))

print("Transposed Array Shape:", transposed_arr.shape)

# Swapaxes: Swapping axis 0 and 1
swapped_arr = np.swapaxes(arr, 0, 1)
print("Swapped Axes Array Shape:", swapped_arr.shape)

# Move axis 0 to the last position
moved_arr = np.moveaxis(arr, 0, -1)
print("Moved Axis Array Shape:", moved_arr.shape)
```

Output:

```mathematica
Transposed Array Shape: (4, 3, 2)
Swapped Axes Array Shape: (3, 2, 4)
Moved Axis Array Shape: (3, 4, 2)
```