

Numpy:

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The predecessor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors. NumPy is a NumFOCUS fiscally sponsored project.

Important of Numpy in python

1. Powerful N-dimensional arrays

Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.

2. Numerical computing tools

NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

3. Open source

Distributed under a liberal BSD license, NumPy is developed and maintained publicly on GitHub by a vibrant, responsive, and diverse community.

4. Interoperable

NumPy supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.

5. Performant

The core of NumPy is well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.

6. Easy to use

NumPy's high level syntax makes it accessible and productive for programmers from any background or experience level.

History:

1. **matrix-sig**

The Python programming language was not originally designed for numerical computing, but attracted the attention of the scientific and engineering community early on.

In 1995 the special interest group (SIG) matrix-sig was founded with the aim of defining an array computing package; among its members was Python designer and maintainer Guido van Rossum, who extended Python's syntax (in particular the indexing syntax) to make array computing easier.

2. **Numeric**

An implementation of a matrix package was completed by Jim Fulton, then generalized by Jim Hugunin and called Numeric (also variously known as the "Numerical Python extensions" or "NumPy"), with influences from the APL family of languages, Basis, MATLAB, FORTRAN, S and S+, and others. Hugunin, a graduate student at the Massachusetts Institute of Technology (MIT), joined the Corporation for National Research Initiatives (CNRI) in 1997 to work on JPython, leaving Paul Dubois of Lawrence Livermore National Laboratory (LLNL) to take over as maintainer. Other early contributors include David Ascher, Konrad Hinsen and Travis Oliphant.

3. **Numarray**

A new package called Numarray was written as a more flexible replacement for Numeric. Like Numeric, it too is now deprecated. Numarray had faster operations for large arrays, but was slower than Numeric on small ones, so for a time both packages were used in parallel for different use cases. The last version of Numeric (v24.2) was released on 11 November 2005, while the last version of numarray (v1.5.2) was released on 24 August 2006.

There was a desire to get Numeric into the Python standard library, but Guido van Rossum decided that the code was not maintainable in its state then.[when?]

4. **NumPy**

In early 2005, NumPy developer Travis Oliphant wanted to unify the community around a single array package and ported Numarray's features to Numeric, releasing the result as NumPy 1.0 in 2006. This new project was part of SciPy. To avoid installing the large SciPy package just to get an array object, this new package was separated and called NumPy. Support for Python 3 was added in 2011 with NumPy version 1.5.0.

In 2011, PyPy started development on an implementation of the NumPy API for PyPy. As of 2023, it is not yet fully compatible with NumPy.

Core features of NumPy

NumPy's core features revolve around efficient data structures and operations for numerical computing. Here's a look at some key features with examples and their benefits:

1. Multidimensional Arrays (ndarrays):

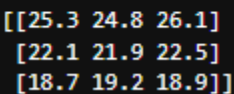
NumPy's core data structure is the ndarray, a powerful n-dimensional array object. Unlike Python lists, ndarrays can hold elements of the same data type, leading to efficient memory usage and faster operations.

Example: Creating a 2D array

```
import numpy as np

data = np.array([[25.3, 24.8, 26.1],
                 [22.1, 21.9, 22.5],
                 [18.7, 19.2, 18.9]])

print(data)
```



```
[[25.3 24.8 26.1]
 [22.1 21.9 22.5]
 [18.7 19.2 18.9]]
```

Ndarrays enable efficient storage and manipulation of large datasets, crucial for scientific data analysis.

2. Broadcasting

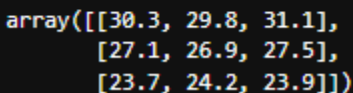
Broadcasting allows performing operations on arrays with different shapes under certain conditions. NumPy automatically expands the smaller array to match the larger one for element-wise operations.

Example: Adding a constant value (5) to each element in the data array:

```
import numpy as np

data = np.array([[25.3, 24.8, 26.1],
                 [22.1, 21.9, 22.5],
                 [18.7, 19.2, 18.9]])

broadcast = data + 5
broadcast
```



```
array([[30.3, 29.8, 31.1],
       [27.1, 26.9, 27.5],
       [23.7, 24.2, 23.9]])
```

Broadcasting simplifies calculations on arrays with slightly different shapes, a common scenario in scientific computing.

3. Vectorized Operations:

NumPy supports applying mathematical functions to entire arrays at once, rather than looping through elements individually. This vectorized approach leverages optimized code for faster computations.

Example: Calculating the average temperature across all time steps for each location (axis=0):

```
# Calculate the mean of each column
column_means = np.mean(data, axis=0)

# Subtract the column means from the respective columns
result = data - column_means

print("Original Data:")
print(data)
print("Columns means")
print(column_means)
print("\nResult after subtracting column means:")
print(result)
```

```
Original Data:
[[25.3 24.8 26.1]
 [22.1 21.9 22.5]
 [18.7 19.2 18.9]]
Columns means
[22.03333333 21.96666667 22.5       ]

Result after subtracting column means:
[[ 3.26666667  2.83333333  3.6       ]
 [ 0.06666667 -0.06666667  0.        ]
 [-3.33333333 -2.76666667 -3.6       ]]
```

Vectorized operations significantly improve performance compared to traditional Python loops, essential for large datasets.

4. Linear Algebra Functions:

NumPy provides a rich set of functions for linear algebra operations like matrix multiplication, solving systems of linear equations, and finding eigenvalues.

Example: Finding the inverse of a square matrix representing a physical system:

```
import numpy as np

data = np.array([[25.3, 24.8, 26.1],
                 [22.1, 21.9, 22.5],
                 [18.7, 19.2, 18.9]])
inverse_matrix = np.linalg.inv(data)
inverse_matrix

array([[ -4.27659574,  7.65957447, -3.21276596],
       [ 0.72340426, -2.34042553,  1.78723404],
       [ 3.4964539 , -5.20094563,  1.41607565]])
```

These functions streamline complex mathematical calculations used in various scientific domains like physics, engineering, and machine learning.

5. Random Number Generation:

NumPy offers functions to generate random numbers with various distributions (normal, uniform, etc.), useful for simulations and statistical analysis.

Example: Generating 20 random numbers from a standard normal distribution:

```
import numpy as np
random_numbers = np.random.normal(size=20)
random_numbers

array([ 0.56417016, -0.17385507,  0.48167783,  0.24692146,  1.00374141,
       -0.35567024,  0.77373402, -0.17698336, -0.82466478, -0.77365317,
        0.55702959, -0.81845747, -0.29203133,  0.51263979,  0.07939904,
        0.0055792 ,  1.0763263 ,  0.4509483 , -0.43525188, -0.10264014])
```

Random number generation is fundamental for creating realistic simulations and performing statistical tests in scientific research.

These core features, along with many others, make NumPy an indispensable tool for scientific computing in Python. By offering efficient data structures, optimized operations, and essential mathematical functions, NumPy empowers scientists and engineers to handle complex computations and analyze data effectively.

Concepts of ndarrays and how it differ from python lists

Ndarrays, short for n-dimensional arrays, are the heart of NumPy's functionality. They offer a powerful and efficient way to store and manipulate numerical data in Python, especially compared to standard Python lists. They can have any number of dimensions, allowing for the representation of scalars, vectors, matrices, or higher-dimensional data structures efficiently.

Here's how ndarrays differ from standard Python lists:

1. Homogeneity:

Ndarrays are homogeneous, meaning all elements within an ndarray must have the same data type (e.g., integers, floats, etc.). This enables efficient storage and operations on large datasets. Python lists, on the other hand, can contain elements of different data types within the same list.

2. Fixed Size:

Ndarrays have a fixed size upon creation. Once created, the size and shape of an ndarray cannot be changed without creating a new array. Lists in Python are dynamic, meaning they can grow or shrink in size dynamically by adding or removing elements.

3. Efficiency:

Ndarrays are implemented in C and optimized for numerical computations, making them much more efficient than Python lists, especially for large datasets. Ndarrays support vectorized operations, where operations are applied element-wise across the entire array, leading to faster computation. Python lists, being implemented in Python itself, may not offer the same level of performance for numerical computations.

4. Multidimensionality:

Ndarrays can have any number of dimensions (1D, 2D, 3D, etc.), allowing for the representation of multi-dimensional data structures like matrices, tensors, or higher-dimensional arrays. Python lists are one-dimensional by default, although you can create lists of lists to represent multi-dimensional data. However, this can be less efficient and less convenient for numerical computations.

5. Functionality:

Ndarrays provide a wide range of mathematical functions and operations optimized for numerical computations, such as element-wise operations, linear algebra operations, statistical operations, and more. While Python lists offer basic list operations and some functional programming methods (e.g., map, filter, reduce), they lack the extensive mathematical functionality provided by NumPy ndarrays.

Overall, ndarrays in NumPy offer a powerful tool for numerical computing and data manipulation, providing efficiency, flexibility, and a rich set of mathematical operations, which sets them apart from standard Python lists.

Time Comparison and Memory Consumption

```
import numpy as np
import time
import sys

python_list = list(range(100000))
numpy_array = np.array(python_list)
# Using Python lists
start_time = time.time()
squared_list = [x**2 for x in python_list]
end_time = time.time()
print("Time taken using Python lists:", end_time - start_time)
print("Memory consumption using Python lists:", sys.getsizeof(squared_list))

# Using NumPy arrays
start_time = time.time()
squared_array = numpy_array ** 2
end_time = time.time()
print("Time taken using NumPy arrays:", end_time - start_time)
print("Memory consumption using NumPy arrays:", squared_array.nbytes)

Time taken using Python lists: 0.021564006805419922
Memory consumption using Python lists: 800984
Time taken using NumPy arrays: 0.0009827613830566406
Memory consumption using NumPy arrays: 400000
```

As you can see, NumPy arrays not only offer better performance but also consume less memory compared to Python lists for the task. This efficiency becomes more pronounced as the size of the data increases. Therefore, for numerical computations and data manipulation tasks, NumPy arrays are often preferred over Python lists.

Universal functions (ufuncs) in NumPy

Universal functions (ufuncs) in NumPy are functions that operate element-wise on ndarrays, performing fast vectorized operations. They are a core component of NumPy and provide efficient computation across arrays of any size and shape. Ufuncs allow you to perform mathematical, logical, bitwise, and other operations on arrays without the need for explicit looping in Python code. This makes them highly efficient for numerical computations, especially when working with large datasets.

| ufuncs | Description | Example | Output |
|-------------|-----------------------------------|-----------------------------------|------------------------------------|
| np.add | Element-wise addition | np.add([1, 2, 3], [4, 5, 6]) | array([5, 7, 9]) |
| np.subtract | Element-wise subtraction | np.subtract([4, 5, 6], [1, 2, 3]) | array([3, 3, 3]) |
| np.multiply | Element-wise multiplication | np.multiply([1, 2, 3], [4, 5, 6]) | array([4, 10, 18]) |
| np.divide | Element-wise division | np.divide([4, 5, 6], [1, 2, 3]) | array([4., 2.5, 2.]) |
| np.power | Element-wise exponentiation | np.power([1, 2, 3], [2, 3, 4]) | array([1, 8, 81]) |
| np.sqrt | Element-wise square root | np.sqrt([4, 9, 16]) | array([2, 3, 4]) |
| np.sin | Element-wise sine function | np.sin(np.pi/2) | 1.0 |
| np.cos | Element-wise cosine function | np.cos(0) | 1.0 |
| np.exp | Element-wise exponential function | np.exp([0, 1, 2]) | array([1., 2.71828183, 7.3890561]) |
| np.log | Element-wise natural logarithm | np.log([1, np.e, np.e**2]) | array([0., 1., 2.,]) |
| np.absolute | Element-wise absolute value | np.absolute([-1, -2, -3]) | array([1, 2, 3]) |

Aggregation in NumPy

Aggregation in NumPy refers to the process of performing a computation across the entire array or along a specified axis to produce a single value that summarizes the data. This summary value can represent various statistical measures such as the mean, median, sum, minimum, maximum, standard deviation, variance, etc.

How it is differ from simple summation and multiplication

Aggregation differs from simple summation or multiplication in that it encompasses a broader range of operations beyond basic arithmetic operations. While summation and multiplication are specific types of aggregation (i.e., computing the sum or product of elements), aggregation also includes other types of operations such as finding the mean, median, minimum, maximum, etc., which provide insights into the distribution and characteristics of the data.

Example:

```
import numpy as np

# Create a sample 2D array
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# Sum of all elements in the array
total_sum = np.sum(arr)
print("Total Sum:", total_sum)

# Mean of all elements in the array
mean_value = np.mean(arr)
print("Mean Value:", mean_value)

# Sum along the rows (axis=1)
sum_along_rows = np.sum(arr, axis=1)
print("Sum Along Rows:", sum_along_rows)

Total Sum: 45
Mean Value: 5.0
Sum Along Rows: [ 6 15 24]
```

In this example, aggregation operations such as `np.sum()`, `np.mean()` are used to compute various statistical measures across the entire array. Additionally, the `axis` parameter allows aggregation along specific axes (rows or columns) of the array, providing flexibility in summarizing multidimensional data.

How NumPy interacts with other python libraries

NumPy interacts seamlessly with other Python libraries, especially those used in data science and machine learning, due to its versatility, efficiency, and extensive ecosystem. This interoperability is crucial for leveraging the strengths of each library and building comprehensive data analysis and machine learning pipelines. Here's how NumPy interacts with other libraries and why this interoperability is important:

1. Integration with SciPy:

SciPy builds on top of NumPy and provides additional functionality for scientific computing. It includes modules for optimization, integration, interpolation, signal processing, linear algebra, and more. NumPy arrays serve as the foundation for many SciPy functions, enabling efficient computation and data manipulation.

2. Matplotlib for Data Visualization:

Matplotlib is a popular library for creating static, interactive, and animated visualizations in Python. Matplotlib accepts NumPy arrays as inputs for plotting data. This seamless integration allows data scientists and researchers to visualize their data effectively for exploratory analysis, model evaluation, and presentation purposes.

3. Pandas for Data Manipulation:

Pandas is a powerful library for data manipulation and analysis, particularly for tabular data. Pandas Series and DataFrames are built on top of NumPy arrays, providing high-level data structures and operations for working with structured data. NumPy arrays can be easily converted to Pandas Series or DataFrames and vice versa, facilitating data preprocessing, cleaning, and transformation tasks.

4. Scikit-learn for Machine Learning:

Scikit-learn is a comprehensive machine learning library that provides efficient implementations of various machine learning algorithms and tools for model selection, evaluation, and preprocessing. Scikit-learn seamlessly integrates with NumPy arrays for data representation and manipulation. NumPy arrays serve as the standard input format for training and testing machine learning models in Scikit-learn.

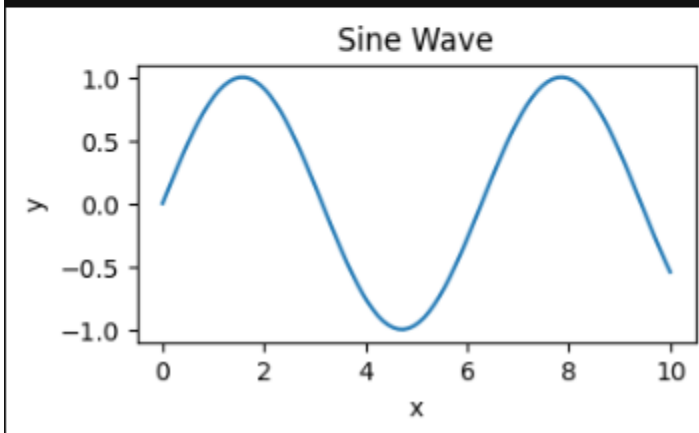
5. Deep Learning Frameworks:

Deep learning frameworks such as TensorFlow and PyTorch often rely on NumPy arrays for data preprocessing and handling. NumPy arrays are compatible with these frameworks and can be easily converted to tensors (the fundamental data structure in TensorFlow and PyTorch) for training neural networks. This interoperability allows data

scientists and researchers to leverage the computational power of deep learning frameworks while benefiting from the flexibility and ease of use of NumPy arrays.

Here's a simple example demonstrating the interoperability between NumPy, Matplotlib, and Pandas:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Generate sample data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Create a Pandas DataFrame
df = pd.DataFrame({'x': x, 'y': y})
# Plot the data using Matplotlib
plt.figure(figsize=(4, 2))
plt.plot(df['x'], df['y'])
plt.title('Sine Wave')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Different NumPy functions

Array Creation

1. np.array()

You can create a NumPy array using the np.array() function.

```
# one dimension
a = np.array([2, 4, 6])

# two dimensional
b = np.array([[3, 2],
              [2, 5]])

array([[3, 2],
       [2, 5]])

# three dimensional
c = np.array([[[2, 5, 2],
               [2, 5, 2],
               [2, 5, 2],
               [3, 5, 2],
               [2, 5, 2]]])
```

2. np.empty()

This function creates an uninitialized array of specified shape and data type. The elements of the array are not initialized, so their values will be whatever happens to already exist at that memory location.

```
np.empty((3, 2))

array([[9.34577196e-307, 9.34598246e-307],
       [1.60218491e-306, 1.69119873e-306],
       [1.24611673e-306, 1.05699581e-307]])
```

3. np.zeros()

This function creates an array filled with zeros of specified shape and data type.

```
import numpy as np
np.zeros((3, 3))

array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

4. `np.ones()`

This function creates an array filled with ones of specified shape and data type.

```
import numpy as np
np.ones((3, 2))

array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

5. `np.arange()`

Generates an array containing evenly spaced values within a specified range.

```
import numpy as np
np.arange(2, 10) # start, stop

array([2, 3, 4, 5, 6, 7, 8, 9])

np.arange(2, 10, 2) # start, stop, step

array([2, 4, 6, 8])
```

6. `np.eye()`

This function returns a 2-D array with ones on the diagonal and zeros elsewhere (identity matrix).

```
import numpy as np
# creates a 3 * 3 identity matrix
np.eye(3, 3)

array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

7. `np.linspace()`

This function returns an array of evenly spaced numbers over a specified interval.

```
import numpy as np
np.linspace(3, 10, 15)

array([ 3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,
        8.5,  9. ,  9.5, 10. ])
```

Array Reshaping

1. `.shape()`:
Refers to the dimensions of an array, represented as a tuple (e.g., (3, 2) for a 3x2 matrix).
2. `Reshape`: Modifies the view of an array by changing its shape without copying the underlying data (if possible). Use it when you want to work with the same data in a different arrangement.

```
arr = np.arange(12)
print("original_array: ", arr)

# .shape()
print("shape:", arr.shape)

# .reshape()
arr.reshape(3, 4) # reshape array to 3 * 4

original_array: [ 0  1  2  3  4  5  6  7  8  9 10 11]
shape: (12,)

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

Broadcasting

Allows performing element-wise operations on arrays with different shapes under specific conditions. NumPy automatically expands the smaller array to match the dimensions of the larger array for calculations.

```

a = np.ones((2, 3)) # 2x3 array of ones
b = np.array([2, 3, 4]) # 1D array

print("shape of a: ", a.shape)
print("shape of b: ", b.shape)

# Element-wise multiplication using broadcasting

result = a * b

print(result)

shape of a: (2, 3)
shape of b: (3,)
[[2. 3. 4.]
 [2. 3. 4.]]

```

Stacking

1. `np.vstack()`
Stacks arrays vertically (along row axis - axis=0).
2. `np.hstack`
Stacks arrays horizontally (along column axis - axis=1).

```

import numpy as np
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

stacked_vertical = np.vstack((array1, array2)) # Vertical stacking
stacked_horizontal = np.hstack((array1, array2)) # Horizontal stacking

print(stacked_vertical)

print(stacked_horizontal)

(2, 3)
[1 2 3 4 5 6]

```

Splitting

1. `np.hsplit()`
Splits an array horizontally (along columns) into a specified number of sub-arrays.

```
import numpy as np
# Creating an example array
arr = np.arange(12).reshape(3, 4)
print("Original Array:")
print(arr)
# Splitting array horizontally into 2 parts
result = np.hsplit(arr, 2)
print("\nAfter splitting horizontally:")
result
```

Original Array:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

After splitting horizontally:

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

2. np.vsplit()

Splits an array vertically (along rows) into a specified number of sub-arrays.

```
import numpy as np
# Creating an example array
arr = np.arange(12).reshape(3, 4)
print("Original Array:")
print(arr)
# Splitting array vertically into 3 parts
result = np.vsplit(arr, 3)
print("\nAfter splitting vertically:")
result
```

Original Array:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

After splitting vertically:

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


3. np.dsplit()

Splits an array along a particular axis (0 for rows, 1 for columns) into a specified number of sub-arrays.

```
import numpy as np
# Creating an example 3D array
arr = np.arange(24).reshape(2, 3, 4)
print("Original 3D Array:")
print(arr)
# Splitting array along depth-wise axis into 2 parts
result = np.dsplit(arr, 2)
print("\nAfter splitting depth-wise:")
print(result)
```

Original 3D Array:

```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]
```

```
   [[12 13 14 15]
    [16 17 18 19]
    [20 21 22 23]]]
```

After splitting depth-wise:

```
[array([[ 0,  1],
        [ 4,  5],
        [ 8,  9]],

       [[12, 13],
        [16, 17],
        [20, 21]]]), array([[ 2,  3],
        [ 6,  7],
        [10, 11],

        [14, 15],
        [18, 19],
        [22, 23]])]
```

Searching

1. np.searchsorted(array, values, side='left', sorter=None)

Finds the indices at which elements from values could be inserted into a sorted array while maintaining order.

side: 'left' (default): returns the index of the first suitable location. 'right' returns the last suitable location.

sorter: Optional pre-computed sorting indices for array.

```
import numpy as np
sorted_array = np.array([1, 3, 5, 7])
search_values = np.array([2, 4, 8])

# Find insertion points for search_values in sorted_array (left side)
insertion_indices = np.searchsorted(sorted_array, search_values)
print(insertion_indices)
```

[1 2 4]

Sorting

1. `np.sort(array, axis=None)`

Sorts an array along a specified axis (default is None - flatten the array). Returns a new sorted array.

2. `np.argsort(array, axis=None)`

Returns the indices that would sort an array along a specified axis.

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

sorted_array = np.sort(arr) # Sort and return a new array
print(sorted_array)

sorting_indices = np.argsort([3, 1, 4, 2]) # Get sorting indices
print(sorting_indices)
```

[1 2 3 4]
[1 3 0 2]

Flattening vs. Raveling:

1. `np.flatten()`

Returns a copy of the array, collapsed into a one-dimensional array (row-major order by default). Modifies the shape attribute.

```
import numpy as np

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

[1 2 3 4 5 6]

2. `np.ravel()`

Similar to `flatten`, but can also return a view of the flattened array if possible (without copying data). Modifies the `shape` attribute.

```
import numpy as np

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

[1 2 3 4 5 6]
```

Shuffling

1. `np.random.shuffle()`

Randomly shuffles the elements of an array in place (modifies the original array).

```
import numpy as np

arr = np.arange(10)
print(arr)
np.random.shuffle(arr)
print(arr)

[0 1 2 3 4 5 6 7 8 9]
[1 9 7 6 3 4 8 2 5 0]
```

Finding Unique Elements

1. `np.unique()`

Returns a sorted array of unique elements from the input array.

```
import numpy as np

arr = np.array([1, 2, 3, 2, 4, 1])
unique_elements = np.unique(arr)
print(unique_elements)

[1 2 3 4]
```

Resizing

1. `np.resize(array, new_shape)`

Returns a new array with the specified `new_shape`. If possible, resizes the original array without copying data. Otherwise, creates a new array.

```

arr = np.arange(6).reshape(2, 3)
print("original_array:\n", arr)
resized_array = np.resize(arr, (3, 2))
print("Resized array:\n", resized_array)

original_array:
[[0 1 2]
 [3 4 5]]
Resized array:
[[0 1]
 [2 3]
 [4 5]]

```

Transpose and Swap Axes:

1. **np.transpose(array)**
Returns a new array with axes interchanged (e.g., rows become columns and vice versa).
2. **np.swapaxes(array, axis1, axis2)**
Swaps two specific axes in the array.

```

matrix = np.arange(12).reshape(3, 4)
print("original_matrix\n", matrix)
transposed_matrix = np.transpose(matrix) # Swap rows and columns
print(transposed_matrix)

swapped_axes = np.swapaxes(matrix, 0, 1) # Swap axis 0 (rows) with axis 1 (columns)
print(swapped_axes)

original_matrix
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]

```