Mastering NumPy

Ali Al Bataineh, Ph.D.

Python's NumPy package has been a lifesaver! It takes care of sorting, reversing algorithms, and much more, saving me so much time and effort. With its high mathematical functions for linear algebra, matrices, and arrays, NumPy is an ideal choice for professionals and newcomers alike in data science and machine learning.

In this tutorial, you'll learn how to install and use NumPy. We'll start with the basics, like creating an array, and then explore more advanced techniques such as querying and data manipulation.

Table of Contents:

1. Introduction to NumPy in Python

NumPy, short for Numerical Python, has become one of my favorite Python libraries. It's widely used in popular packages such as pandas, SciPy, Matplotlib, and more. Arrays in NumPy are naturally faster than Python lists, which has made a significant difference in optimizing my computational performance. From simple math calculations to complex data science operations, NumPy has been an invaluable tool in my workflow.

2. Steps to Install NumPy in Your Python Environment

Installing NumPy in your Python environment is a simple process. Here are the steps to install NumPy using package managers such as pip or conda:

1. Using pip (Python Package Installer): For users who have Python installed on their system, pip is

   the most common package manager. To install NumPy using pip, open a terminal or command prompt and run the following command:**pip install numpy**

1. Using conda (Anaconda or Miniconda): For those using Anaconda or Miniconda, the conda package manager is available. To install NumPy using conda, open the Anaconda Prompt or terminal and enter the following command:**conda install numpy**

After running the appropriate command, the package manager will download and install NumPy in your Python environment. Once the installation is complete, you can import NumPy in your Python scripts or Jupyter Notebooks using the following line of code: **import numpy as np**

By convention, NumPy is imported as np for shorter and more convenient usage. Now you're ready to start using NumPy for various numerical operations and data manipulation tasks.

4. Creating NumPy Arrays: A Step-by-Step Guide

Creating NumPy arrays is quite straightforward. In this step-by-step guide, I'll show you various methods to create NumPy arrays:

Import the NumPy library

First, you need to import the NumPy library. By convention, it is imported as np:

In [5]:
```
import numpy as np
```

Create a 1-dimensional (1D) array

To create a 1D array, pass a Python list as an argument to the np.array() function:

In [6]:
```
arr_1d = np.array([1, 2, 3, 4, 5])
print(arr_1d)
```

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

Create a 2-dimensional (2D) array

To create a 2D array, pass a list of lists as an argument to the np.array() function:

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

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

Create arrays with specific data types

By default, NumPy infers the data type of the array from its elements. However, you can explicitly specify the data type using the dtype parameter:

In [8]:
```
arr_float = np.array([1, 2, 3, 4, 5], dtype=float)
print(arr_float)
```

```
[1. 2. 3. 4. 5.]
```

Create arrays with predefined functions

NumPy provides several functions to create arrays with specific characteristics, such as zeros, ones, or a range of values:

np.zeros()

In [9]:
```
#Create a 1D array filled with zeros.
zeros_arr = np.zeros(5)
print(zeros_arr)
```

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

In [12]:
```
#Create a 2D array filled with zeros.
zeros_arr2d = np.zeros(shape=(2,3))
print(zeros_arr2d)
```

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

np.ones()

In [13]:
```
#Create a 1D array filled with ones.
ones_arr = np.ones(5)
print(ones_arr)
```

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

In [14]:
```
#Create a 2D array filled with ones.
ones_arr2d = np.ones(shape=(3,4))
print(ones_arr2d)
```

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

np.arange()

In [15]:
```
#Create an array with a range of values.
range_arr = np.arange(0, 10, 2)
print(range_arr)
```

```
[0 2 4 6 8]
```

np.linspace()

In [16]:
```
#Create an array with evenly spaced values between a specified range.
linspace_arr = np.linspace(0, 1, 5)
print(linspace_arr)
```

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

np.empty()

In [17]:

```
#Create an array with random values, depending on the memory state.
empty_arr = np.empty((3, 4))
print(empty_arr)
```

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

np.vstack()

In [20]:

```
#Stack arrays vertically
import numpy as np

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

# Stack arrays vertically
vstacked_arr = np.vstack((arr1, arr2))
print(vstacked_arr)
```

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

np.hstack()

In [21]:

```
#Stack arrays horizontally
import numpy as np

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

# Stack arrays horizontally
hstacked_arr = np.hstack((arr1, arr2))
print(hstacked_arr)
```

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

np.eye()

In [23]:

```
# # function in NumPy is used to create a 2D identity matrix with a specified size.
# An identity matrix is a square matrix with ones on its diagonal and zeros elsewhere.
identity_matrix = np.eye(4)
print(identity_matrix)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

4. Practical Examples: Utilizing NumPy

In [24]:
```
#Calculating the element-wise sum of two arrays
import numpy as np

arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([6, 7, 8, 9, 10])
sum_arr = arr1 + arr2
print(sum_arr)
```

```
[ 7  9 11 13 15]
```

In [30]:
```
#Element-wise subtraction between two arrays
arr = np.array([10, 20, 30, 40, 50])
scalar = 5
subtraction_result = arr - scalar
print(subtraction_result)
```

```
[ 5 15 25 35 45]
```

In [25]:
```
#Computing the dot product of two vectors using np.dot()
import numpy as np

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

```
32
```

In [27]:
```
#Matrix multiplication using np.matmul()
import numpy as np

# Define two matrices A and B
A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = np.array([[7, 8],
              [9, 10],
              [11, 12]])

# Matrix multiplication
matmul_result = np.matmul(A, B)
print(matmul_result)
```

```
[[ 58  64]
 [139 154]]
```

In [28]:
```
#Element-wise division between two arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([5, 4, 3, 2, 1])
division_result = arr1 / arr2
print(division_result)
```

```
[0.2 0.5 1.  2.  5. ]
```

In [29]:
```
#Dividing each element of an array by a scalar value
arr = np.array([2, 4, 6, 8, 10])
scalar = 2
division_result = arr / scalar
print(division_result)
```

```
[1. 2. 3. 4. 5.]
```

In [33]:
```
#Reshaping an array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
reshaped_arr = arr.reshape(3, 3)
print("Before Reshape:\n", arr)
print("After Reshape: \n", reshaped_arr)
```

```
Before Reshape:
 [1 2 3 4 5 6 7 8 9]
After Reshape:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

In [35]:
```
#Transposing a matrix
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
transposed_matrix = matrix.T
print(transposed_matrix)
```

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

In [36]:
```
# Calculating the inverse of a matrix
matrix = np.array([[1, 2], [3, 4]])
inverse_matrix = np.linalg.inv(matrix)
print(inverse_matrix)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Basic Statistics and Operations with NumPy Arrays

In [37]:
```
arr = np.array([4, 2, 9, 7, 5])

# Get the minimum value in the array
min_value = np.min(arr)
print("Minimum value:", min_value)

# Get the maximum value in the array
max_value = np.max(arr)
print("Maximum value:", max_value)

# Calculate the standard deviation of the array
std_dev = np.std(arr)
print("Standard deviation:", std_dev)

# Calculate the mean (average) value of the array
mean_value = np.mean(arr)
```

```
print("Mean value:", mean_value)

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

# Find the index of the minimum value in the array
min_index = np.argmin(arr)
print("Index of minimum value:", min_index)

# Find the index of the maximum value in the array
max_index = np.argmax(arr)
print("Index of maximum value:", max_index)

# Compute the cumulative sum of the array elements
cumulative_sum = np.cumsum(arr)
print("Cumulative sum:", cumulative_sum)

# Compute the element-wise square root of the array
sqrt_arr = np.sqrt(arr)
print("Square root of array elements:", sqrt_arr)

# Compute the median of the array elements
median_value = np.median(arr)
print("Median value:", median_value)

# Sort the array
sorted_arr = np.sort(arr)
print("Sorted array:", sorted_arr)

# Calculate the element-wise exponential of the array
exp_arr = np.exp(arr)
print("Exponential of array elements:", exp_arr)
```

```
Minimum value: 2
Maximum value: 9
Standard deviation: 2.4166091947189146
Mean value: 5.4
Total sum: 27
Index of minimum value: 1
Index of maximum value: 2
Cumulative sum: [ 4  6 15 22 27]
Square root of array elements: [2.         1.41421356 3.         2.64575131 2.23606798]
Median value: 5.0
Sorted array: [2 4 5 7 9]
Exponential of array elements: [5.45981500e+01 7.38905610e+00 8.10308393e+03 1.09663316e
+03
 1.48413159e+02]
```

slicing and indexing with NumPy arrays

In [38]:

```
#Basic Indexing
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
element = arr[1, 2]  # Access the element at the second row and third column
print(element)
```

```
6
```

In [39]:
```
#Slice notation
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
sub_arr = arr[1:5]  # Access the elements from index 1 (inclusive) to 5 (exclusive)
print(sub_arr)
```

```
[2 3 4 5]
```

In [40]:
```
#Strides in slicing
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
every_second_element = arr[1::2]  # Access every second element starting from index 1
print(every_second_element)
```

```
[2 4 6 8]
```

In [41]:
```
#Multi-dimensional slicing
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
sub_arr = arr[1:, :2]  # Access all rows from index 1 onwards and columns up to index 2
print(sub_arr)
```

```
[[4 5]
 [7 8]]
```

In [50]:
```
# Access the last column of the 2D array
# Create a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Access the last column of the 2D array
last_column = arr[:, -1]

print(last_column)

# Access the last two columns of the 2D array
last_two_columns = arr[:, -2:] #selects all rows and the last two columns of the 2D arr

print(last_two_columns)
```

```
[3 6 9]
[[2 3]
 [5 6]
 [8 9]]
```

In [42]:
```
#Using slices to modify elements
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
arr[1:5] = -1  # Set the elements from index 1 (inclusive) to 5 (exclusive) to -1
print(arr)
```

```
[ 1 -1 -1 -1 -1  6  7  8  9]
```

In [43]:
```
#Boolean indexing
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
even_elements = arr[arr % 2 == 0]  # Access all even elements in the array
print(even_elements)
```

```
[2 4 6 8]
```

In [45]:
```
#concatenate
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
concatenated_arr = np.concatenate((arr1, arr2))
print(concatenated_arr)
```

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

In [46]:
```
#split
arr = np.array([1, 2, 3, 4, 5, 6])
split_arr = np.split(arr, 2)
print(split_arr)
```

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

In [47]:
```
#tile
arr = np.array([1, 2, 3])
tiled_arr = np.tile(arr, 3)
print(tiled_arr)
```

```
[1 2 3 1 2 3 1 2 3]
```

In [48]:
```
#repeat
arr = np.array([1, 2, 3])
repeated_arr = np.repeat(arr, 2)
print(repeated_arr)
```

```
[1 1 2 2 3 3]
```

In [51]:
```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Shape of the array
arr_shape = arr.shape
print("Shape of the array:", arr_shape)

# Size of the array
arr_size = arr.size
print("Size of the array:", arr_size)

# Number of dimensions of the array
arr_ndim = arr.ndim
print("Number of dimensions of the array:", arr_ndim)

# Length of each dimension of the array
arr_length_0 = len(arr)
arr_length_1 = len(arr[0])
print("Length of dimension 0:", arr_length_0)
print("Length of dimension 1:", arr_length_1)

# Number of rows and columns
num_rows = arr_shape[0]
num_columns = arr_shape[1]
print("Number of rows:", num_rows)
print("Number of columns:", num_columns)
```

```
Shape of the array: (3, 3)
Size of the array: 9
Number of dimensions of the array: 2
Length of dimension 0: 3
Length of dimension 1: 3
Number of rows: 3
Number of columns: 3
```

np.where: Return an array with elements from value if true where the corresponding element in logical condition is True, and value if false where it is False.

In [52]:
```
arr = np.array([1, 2, 3, 4, 5, 6])
arr_new = np.where(arr % 2 == 0, 0, arr)

print("New array with even values replaced by 0:")
print(arr_new)
```

```
New array with even values replaced by 0:
[1 0 3 0 5 0]
```

In [53]:
```
arr = np.array([1, 2, 3, 4, 5, 6])
arr_size = arr.size

print("Size of the array:", arr_size)
```

```
Size of the array: 6
```

In [54]:
```
arr = np.array([1, 2, 3, 4, 5, 6])
arr.resize((2, 3))

print("Resized array:")
print(arr)
```

```
Resized array:
[[1 2 3]
 [4 5 6]]
```

reshape returns a new array with the same data as the original array but with a new shape. The original array is not modified.

resize, on the other hand, changes the shape of the array in place, modifying the original array.

Summary

NumPy is an essential tool for anyone working in data science or scientific computing. With its powerful array processing capabilities and vast library of mathematical functions, NumPy can help you tackle complex data problems with ease. In this tutorial, we have covered the basics of NumPy and explored some of its advanced features. We have seen how to install NumPy, create arrays, and perform basic operations such as indexing and slicing.

We have also looked at some more advanced techniques, such as performing queries and data manipulation, and how to use NumPy for statistical operations. By utilizing NumPy, data scientists

and machine learning engineers can optimize computational performance in their workflow - from simple mathematical calculations to complex data manipulation for data science operations.