Introduction to NumPy in Data Science

A Fundamental Python Library for Numerical Computing

Overview

- Introduction to NumPy
- Why NumPy is Essential for Data Science
- Installation: How to Install NumPy

Overview

- Introduction to NumPy

  - NumPy is a Python library used for working with arrays.
  - It also has functions for working in domain of linear algebra, Fourier transform, and matrices.
  - NumPy was created in 2005 by Travis Oliphant. It is an opensource project and you can use it freely.
  - NumPy stands for Numerical Python.

Overview

- Why NumPy is Essential for Data Science

1. Efficient Array Operations: NumPy provides highly efficient and optimized array operations.
2. Multi-dimensional Arrays: easy to represent complex data structures like matrices and tensors.
3. Mathematical and Statistical Functions.
4. Broadcasting: allows for element-wise operations between arrays of different shapes, making it easy to work with data of varying dimensions. This simplifies operations like adding a scalar to an entire array or combining arrays with different shapes.

Overview

- Why NumPy is Essential for Data Science

5. Random Number Generation: NumPy provides tools for generating random numbers from various probability distributions, allowing data scientists to simulate and analyze data with randomness.
6. Integration with Other Libraries: pandas, Matplotlib, and scikit-learn.
7. Data Storage and Loading: NumPy arrays can be easily saved to and loaded from disk, allowing data scientists to store and share data efficiently. The standardization of NumPy arrays simplifies data exchange between different data science tools and platforms.

Overview

- Introduction to NumPy
- Why NumPy is Essential for Data Science
- Installation: How to Install NumPy

  - Numpy can be installed for **Mac** and **Linux** users via the following pip command:
  - pip install numpy

What is NumPy?

- Definition: Numerical Python (NumPy)
- Core Library for Scientific and Numeric Computing in Python
- Provides Support for Large, Multi-dimensional Arrays and Matrices
- Foundation for Data Science and Machine Learning

NumPy Features

- Multi-dimensional Arrays
- Mathematical Functions
- Random Number Generation
- Broadcasting
- Integration with Other Libraries (e.g., pandas, matplotlib)

Importing NumPy

- Importing NumPy into Python
- Standard Convention: import numpy as np
- Now You Can Use np as a Shortcut

NumPy Arrays

- Creating NumPy Arrays
- Array Properties: Shape, Size, Data Type

NumPy Arrays

- Creating NumPy Arrays

  - What?
    - In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is **array.**
  - Why?
    - They provide several advantages over Python's built-in lists when it comes to numerical computing and data science.
  - How?
    - There are a number of ways to initialize new numpy arrays, for example from

    1. a Python list or tuples
    2. using functions that are dedicated to generating numpy arrays, such as arange, linspace, etc.
    3. reading data from files

NumPy Arrays

- Array Properties:
  - Shape,
  - Size,
  - Data Type

```python
import numpy as np

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

# Accessing Array Properties
array_shape = my_array.shape  # Shape of the array (dimensions)
array_size = my_array.size    # Size of the array (total number of elements)
array_dtype = my_array.dtype  # Data type of the elements in the array

# Printing Array Properties
print("Array:")
print(my_array)
print("\nArray Shape:", array_shape)
print("Array Size:", array_size)
print("Array Data Type:", array_dtype)
```

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

Array Shape: (3, 3)
Array Size: 9
Array Data Type: int64
```

NumPy Arrays

- Using array-generating functions

  - Arange

    x = arange(0, 10, 1) # create a range, arguments: (start, stop, step)
  - linspace and logspace
  - mgrid

    Create a grid of coordinates using mgrid, grid is now a multi-dimensional array containing coordinate grids for two dimensions.
  - diag
  - random data
  - zeros and ones

1. Using arange(start, stop, step)

This function generates a sequence of numbers, starting from start, ending at stop (exclusive), with a step size of step.

```
import numpy as np

# Example: Generate numbers from 0 to 9
x = np.arange(0, 10, 1)
print(x)
```

You will get arange example: [0 1 2 3 4 5 6 7 8 9]

2. Using linspace(start, stop, num)

This generates num evenly spaced values between start and stop (inclusive).

```
# Example: Generate 5 evenly spaced numbers from 0 to 10
y = np.linspace(0, 10, 5)
print(y)
```

You will get Linspace example: [0 2.5 5 7.5 10]

3. Using logspace(start, stop, num)

This generates num logarithmically spaced values between powers of 10.

```
# Example: Generate 4 logarithmically spaced numbers between 10^1 and 10^3
z = np.logspace(1, 3, 4)
print(z)
```

You will get Logspace example: [10.   46.4159   215.4435   1000.]

4. Using mgrid for Mesh Grid

This creates a multi-dimensional mesh grid for two dimensions. It’s useful for generating coordinate grids.

```
# Example: Create a 2D grid for coordinates
X, Y = np.mgrid[0:3, 0:2]
print("Mgrid example (X):", X)
print("Mgrid example (Y):", Y)
```

```
Mgrid example (X):
[[0 0]
 [1 1]
 [2 2]]
Mgrid example (Y):
[[0 1]
 [0 1]
 [0 1]]
```

5. Using diag() for Diagonal Matrices

This extracts the diagonal of a matrix or creates a diagonal matrix.

```
# Example: Create a diagonal matrix
diag_matrix = np.diag([1, 2, 3])
print("Diagonal matrix example:", diag_matrix)
```

```
Diagonal matrix example:
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

6. Generating Random Data with random

Here are a few ways to generate random arrays.

```
# Example: Create a 2x3 matrix of random values between 0 and 1
rand_array = np.random.rand(2, 3)
print("Random array example:", rand_array)

# Example: Create a 2x3 matrix of normally distributed random numbers
randn_array = np.random.randn(2, 3)
print("Random normal distribution example:", randn_array)
```

```
Random array example:
[[0.11697915 0.64027738 0.53711974]
 [0.5296738  0.23264256 0.8197505 ]]

Random normal distribution example:
[[ 0.34674895  0.15821207 -1.17543321]
 [-0.24928552 -0.24063729  0.94453738]]
```

7. Creating Arrays of Zeros and Ones with zeros() and ones()

These functions create arrays filled with zeros or ones, respectively.

```
# Example: Create a 2x3 array filled with zeros
zeros_array = np.zeros((2, 3))
print("Zeros array example:", zeros_array)

# Example: Create a 2x3 array filled with ones
ones_array = np.ones((2, 3))
print("Ones array example:", ones_array)
```

```
Zeros array example:
[[0. 0. 0.]
 [0. 0. 0.]]

Ones array example:
[[1. 1. 1.]
 [1. 1. 1.]]
```

Additional Example - eye(): Identity Matrix

You can also use eye() to create an identity matrix.

```
# Example: Create a 3x3 identity matrix
identity_matrix = np.eye(3)
print("Identity matrix example:", identity_matrix)
```

```
Identity matrix example:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Indexing and Slicing

- Accessing Elements in an Array
- Slicing: Extracting Subarrays
- Indexing with Boolean Masks

    - Read all examples in Github

Array Operations

- Element-Wise Operations
- Array Addition, Subtraction, Multiplication, and Division
- Broadcasting: Working with Arrays of Different Shapes

Array Operations

- Element-Wise Operations

  When we add, subtract, multiply and divide arrays with each other, the default behavior is element-wise operations.

- Array Addition, Subtraction, Multiplication, and Division

```python
import numpy as np

# Create two NumPy arrays
array1 = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

array2 = np.array([[9, 8, 7],
                   [6, 5, 4],
                   [3, 2, 1]])

# Perform Array Operations
array_addition = array1 + array2
array_subtraction = array1 - array2
array_multiplication = array1 * array2
array_division = array1 / array2

# Printing Results
print("Array 1:")
print(array1)
print("\nArray 2:")
print(array2)

print("\nArray Addition:")
print(array_addition)

print("\nArray Subtraction:")
print(array_subtraction)

print("\nArray Multiplication:")
print(array_multiplication)

print("\nArray Division:")
print(array_division)
```

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

Array 2:
[[9 8 7]
 [6 5 4]
 [3 2 1]]

Array Addition:
[[10 10 10]
 [10 10 10]
 [10 10 10]]

Array Subtraction:
[[-8 -6 -4]
 [-2  0  2]
 [ 4  6  8]]

Array Multiplication:
[[ 9 16 21]
 [24 25 24]
 [21 16  9]]

Array Division:
[[0.11111111 0.25       0.42857143]
 [0.66666667 1.         1.5       ]
 [2.33333333 4.         9.        ]]
```

- Element-wise multiplication VS

```python
import numpy as np

# Generate two random 5x5 matrices
matrix1 = np.random.rand(5, 5)
matrix2 = np.random.rand(5, 5)

# Perform matrix multiplication
result_matrix1 = np.dot(matrix1, matrix2)

# Alternatively, you can use the @ operator for
result_matrix2 = matrix1 @ matrix2

# Perform element-wise multiplication
result_matrix3 = matrix1 * matrix2

print("Matrix 1:")
print(matrix1)
print("\nMatrix 2:")
print(matrix2)
print("\nResult Matrix1:")
print(result_matrix1)
print("\nResult Matrix2:")
print(result_matrix2)
print("\nResult Matrix3:")
print(result_matrix3)
```

```
Matrix 1:
[[0.11375134 0.59385753 0.95056089 0.88470112 0.36920745]
 [0.70330434 0.47158135 0.50950612 0.39875395 0.8506669 ]
 [0.28143272 0.51721639 0.90002967 0.80633582 0.95898317]
 [0.78550522 0.44556475 0.87063692 0.15821857 0.9176677 ]
 [0.39616018 0.49859386 0.79323223 0.24351393 0.37399191]]

Matrix 2:
[[0.25197668 0.85783221 0.51405033 0.64106111 0.4893183 ]
 [0.14552214 0.45185413 0.6776783  0.41929789 0.31264726]
 [0.0283149  0.21141357 0.84100064 0.12170041 0.40641155]
 [0.87361665 0.40741922 0.80114702 0.99388779 0.85181818]
 [0.19169379 0.37712256 0.291272   0.97894457 0.9583435 ]]

Result Matrix1:
[[0.98566155 1.06655873 2.07665606 1.67833569 1.73507953]
 [0.77169409 1.40738532 1.67684523 1.93967365 1.85354442]
 [1.05992469 1.35557782 2.17742093 2.24701598 2.27108572]
 [0.60155411 1.4697609  1.83199223 1.85193539 1.89171791]
 [0.47926962 0.97308376 1.51266515 1.1677018  1.23795348]]

Result Matrix2:
[[0.98566155 1.06655873 2.07665606 1.67833569 1.73507953]
 [0.77169409 1.40738532 1.67684523 1.93967365 1.85354442]
 [1.05992469 1.35557782 2.17742093 2.24701598 2.27108572]
 [0.60155411 1.4697609  1.83199223 1.85193539 1.89171791]
 [0.47926962 0.97308376 1.51266515 1.1677018  1.23795348]]

Result Matrix3:
[[0.02866269 0.50943012 0.48863614 0.56714749 0.18065996]
 [0.10234635 0.21308598 0.34528124 0.16719669 0.26595867]
 [0.00796874 0.10934657 0.75692553 0.0981314  0.38974184]
 [0.68623044 0.18153164 0.69750818 0.1572515  0.78168604]
 [0.07594145 0.188031   0.23104634 0.23838664 0.35841272]]
```

Universal Functions (UFuncs)

- Overview of UFuncs
- Mathematical Functions
- Aggregation Functions (e.g., mean, sum, max)
- Element-wise Functions

Universal Functions (UFuncs)

- Overview of Ufuncs
  - UFuncs are a class of functions that operate element-wise on arrays, meaning they apply a specific operation to each element of an array individually, producing a new array as the result.

- Mathematical Functions

  NumPy provides a wide range of mathematical UFuncs. Examples:
  - np.add(): Element-wise addition
  - np.subtract(): Element-wise subtraction
  - np.multiply(): Element-wise multiplication
  - np.divide(): Element-wise division
  - np.sqrt(): Element-wise square root
  - np.sin(), np.cos(), np.tan(): Trigonometric functions
  - np.exp(): Exponential function
  - np.log(), np.log10(): Logarithmic functions

Universal Functions (UFuncs)

- Aggregation Functions (e.g., mean, sum, max)

  Aggregation UFuncs perform operations that aggregate data. Example:

  - np.sum(): Sum of array elements
  - np.mean(): Mean (average) of array elements
  - np.max(): Maximum element
  - np.min(): Minimum element
  - np.std(): Standard deviation
  - np.var(): Variance
  - np.median(): Median

Universal Functions (UFuncs)

- Element-wise Functions

  Element-wise UFuncs apply operations element by element. Examples

  - np.abs(): Absolute values
  - np.round(): Round to nearest integer
  - np.floor(): Floor (round down)
  - np.ceil(): Ceiling (round up)
  - np.sign(): Sign function (-1, 0, 1)
  - np.isnan(): Check for NaN (Not-a-Number)
  - np.isinf(): Check for infinity

Random Number Generation

- Generating Random Numbers
- Common Probability Distributions (e.g., uniform, normal)
- Setting the Random Seed for Reproducibility

Random Number Generation

- Generating Random Numbers
  - Use NumPy's random module for random number generation.
    - Example: Generate a random integer between 1 and 10.

```
import numpy as np
random_number = np.random.randint(1, 11)
```

Example: Generate a random matrix.

```
import numpy as np

# Generate two random 5x5 matrices
matrix1 = np.random.rand(5, 5)
matrix2 = np.random.rand(5, 5)
```

Random Number Generation

- Common Probability Distributions (e.g., uniform, normal)
  NumPy provides functions for various probability distributions.

Uniform Distribution

python

```python
uniform_data = np.random.uniform(0, 1, 1000)
```

Normal Distribution

python

```python
normal_data = np.random.normal(0, 1, 1000)
```

Random Number Generation

- Setting the Random Seed for Reproducibility
  - **Random Number Generator (RNG):** NumPy includes a pseudo-random number generator (PRNG) that generates random numbers. This PRNG is deterministic, meaning that given the same initial conditions, it will produce the same sequence of random numbers every time.
  - **Random Seed:** The "seed" is the initial value or starting point for the random number generator. When you set the seed, you specify the starting point for the random number sequence.
  - **Reproducibility:** By setting the random seed to a specific value, such as 42 in np.random.seed(42), you ensure that the random number generator starts from the same initial state every time you run your code. As a result, it produces the same sequence of random numbers in each run.

Random Number Generation

- Setting the Random Seed for Reproducibility

```python
import numpy as np

# Set the random seed to 42
np.random.seed(42)

# Generate random numbers
random_numbers1 = np.random.rand(5)
print(random_numbers1)

# Reset the seed to 42 and generate again
np.random.seed(42)
random_numbers2 = np.random.rand(5)
print(random_numbers2)
```

```
[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]
[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]
```

Exercise: Generating and Manipulating a Random 4x4 Matrix

1. Generate a 4x4 matrix of random integers between 1 and 100. `np.random.randint(1, 101, size=(4, 4))`
2. Calculate the sum of all elements in the matrix.
3. Find the maximum and minimum values in the matrix.
4. Replace all even numbers in the matrix with 0. `matrix[matrix % 2 == 0] = 0`
5. Find the row-wise sum of the modified matrix. `np.sum(matrix, axis=1)`
6. Transpose the matrix (swap rows and columns).

Broadcasting: Working with Arrays of Different Shapes

Broadcasting Rules:

NumPy follows specific rules when broadcasting arrays:

1. **Dimension Compatibility:** Two dimensions are compatible when either:
    1. They are equal, or
    2. One of them is 1.
2. **Size Compatibility:** For broadcasting to work, the size of the dimension must be either the same or one of them must be 1. In other words, the smaller array can be "broadcasted" to match the shape of the larger array.

- **Example 1: Adding Scalars to Arrays:**

You can add a scalar to an array, and NumPy will broadcast the scalar to all elements of the array.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
result = arr + 10  # Broadcasting: Add 10 to all elements
print(result)
```

```
[11 12 13 14 15]
```

- **Example 2: Broadcasting with 1D and 2D Arrays:**

You can even broadcast a 1D array to a 2D array

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([[10, 20, 30], [100, 200, 300]])
result = arr1 + arr2  # Broadcasting: Add arr1 to each row of arr2
print(result)
```

```
[[ 11  22  33]
 [101 202 303]]
```

- **Example 3: Broadcasting with 1D and 2D Arrays (Different Sizes):**

In this example, the dimensions are compatible because one of them is 1, and broadcasting still works.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([[10], [100]])
result = arr1 + arr2  # Broadcasting: Add arr1 to each element of arr2
print(result)
```

```
[[ 11  12  13]
 [101 102 103]]
```

- **Example 4: Broadcasting Error: Incompatible Shapes:**

If the dimensions are not compatible, broadcasting will result in an error.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([[10, 20], [100, 200]])
result = arr1 + arr2  # Broadcasting error: Incompatible shapes
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/12/y3f8dw5d3dsc_5ynrf2qky680000gn/T/ipykernel_53912/3105536372.py in <module>
      1 arr1 = np.array([1, 2, 3])
      2 arr2 = np.array([[10, 20], [100, 200]])
----> 3 result = arr1 + arr2  # Broadcasting error: Incompatible shapes

ValueError: operands could not be broadcast together with shapes (3,) (2,2)
```

Practical Examples

- Basic Data Analysis with NumPy
- Loading and Manipulating Data
- Statistical Analysis with NumPy Functions

1. Loading Data with NumPy

- Text files (np.loadtxt())
- CSV files (np.genfromtxt())
- Binary files (np.fromfile())

2. Manipulating Data

- Array slicing and indexing
- Element-wise operations
- Filtering and masking

3. Statistical Analysis with NumPy

- Descriptive statistics (mean, median, std, etc.)
- Aggregation functions (sum, min, max)
- Correlation and covariance

4. Descriptive Statistics

- Mean: np.mean()
- Median: np.median()
- Standard Deviation: np.std()

5. Aggregation Functions

- Sum: np.sum()
- Minimum: np.min()
- Maximum: np.max()

6. Correlation and Covariance

- np.corrcoef(): Correlation coefficient matrix
- np.cov(): Covariance matrix

7. Summary

Integration with pandas

- Combining NumPy and pandas for Data Manipulation

  - NumPy provides the efficiency and flexibility for mathematical operations on data.
  - pandas provides high-level data manipulation tools, data alignment, and built-in handling of missing data.
  - By combining NumPy and pandas, you can take advantage of both libraries' strengths to efficiently manipulate, analyze, and visualize structured data in various data science and analysis tasks. This combination is a common and powerful approach in the Python data ecosystem.

Conclusion

- The Importance of NumPy in Data Science
- Encourage Further Exploration and Practice

Exercise: Generating and Manipulating a Random 4x4 Matrix

1. Generate a 4x4 matrix of random integers between 1 and 100.
2. Calculate the sum of all elements in the matrix.
3. Find the maximum and minimum values in the matrix.
4. Replace all even numbers in the matrix with 0.
5. Find the row-wise sum of the modified matrix.
6. Transpose the matrix (swap rows and columns).

Exercise: Broadcasting Operations on the Matrix

7. Add a constant value (5) to each element of the matrix using broadcasting.
8. Subtract a vector [1, 2, 3, 4] from each row of the matrix
9. Multiply a column vector [1, 2, 3, 4] with each column of the matrix
10. Subtract the mean of each row from the matrix