NumPy for Data Science - part 1

Type	Data science masterclass

I. Introduction to NumPy

What is NumPy?

NumPy (short for Numerical Python) is a library in Python that helps you work with numbers and arrays (which are like lists but with special abilities). It's a tool that allows you to do fast and efficient mathematical operations, like adding, multiplying, or calculating statistics, especially when you're working with lots of data.

Why is NumPy Essential for Data Science?

In Data Science, you often need to handle large amounts of data and perform complex calculations. NumPy is crucial for the following reasons:

- **Faster calculations**: NumPy can process large data much faster than normal Python lists.
- **Efficient memory usage**: It uses less memory, making it better for handling big datasets.
- **Supports complex math**: NumPy has built-in functions for things like adding numbers, finding averages, or even solving equations without writing complicated code.
- **Works well with other libraries**: Many other data science libraries (like pandas, scikit-learn, etc.) use NumPy arrays to make their work faster and easier.

Installing and Importing NumPy

To start using NumPy, you first need to install it. If you haven't already, you can do it by running this command in your terminal or command prompt:

```
pip install numpy
```

After installation, you can import NumPy into your Python code like this:

```
import numpy as np
```

We use `np` as a shortcut to make the code shorter and easier to write.

NumPy Arrays vs Python Lists

In Python, you might be familiar with lists, which are a way to store multiple values (like numbers, words, etc.) in a single variable. But NumPy introduces **arrays**, which are like lists but with more powerful features.

Key Differences:

1. **Speed**:
    - **Python List**: A normal Python list can store any type of data (numbers, words, etc.), but it is slower when you need to do math on the numbers in the list.
    - **NumPy Array**: A NumPy array is faster when you want to do math because it is designed specifically for numerical data.
2. **Memory**:
    - **Python List**: Stores data in a flexible way but uses more memory.
    - **NumPy Array**: Stores numbers in a more memory-efficient way, which is helpful when working with large datasets.
3. **Operations**:
    - **Python List**: If you want to do something like multiply all the numbers by 2, you'd need a loop or a list comprehension.
    - **NumPy Array**: You can do things like this directly with NumPy without needing a loop.

Example:

Let's say you want to square the numbers in a list. Here's how you could do it with a Python list and a NumPy array:

```
# Using Python List
py_list = [1, 2, 3, 4, 5]
squared_list = [x ** 2 for x in py_list]  # Using list comprehension

# Using NumPy Array
import numpy as np
np_array = np.array([1, 2, 3, 4, 5])
squared_array = np_array ** 2  # This is much faster and easier!

print("Squared List:", squared_list)
print("Squared NumPy Array:", squared_array)
```

II. Creating NumPy Arrays

In this section, we will learn how to create NumPy arrays using various methods. NumPy arrays are a powerful way to store and manipulate data, and knowing how to create them from different sources is essential for working with data in Python.

Creating Arrays from Lists and Tuples

The most common way to create a NumPy array is by converting a Python list or tuple into an array using `numpy.array()`.

- **From a list**:

```
import numpy as np
py_list = [1, 2, 3, 4, 5]
np_array_from_list = np.array(py_list)
print(np_array_from_list)
```

- **From a tuple**:

```
py_tuple = (1, 2, 3, 4, 5)
np_array_from_tuple = np.array(py_tuple)
print(np_array_from_tuple)
```

Both methods convert a list or tuple into a NumPy array. The main advantage of this is that you can now perform operations on the array much faster than you could on a list or tuple.

Using `numpy.array()`

The `numpy.array()` function is the general method to create an array from a list, tuple, or another array. You can also specify the type of elements in the array by using the `dtype` argument.

Example:

```
arr = np.array([1, 2, 3, 4], dtype=float)  # Creates an array of floats
print(arr)
```

Using `numpy.arange()`

`numpy.arange()` creates an array with evenly spaced values within a specified range. It is similar to Python's built-in `range()` function, but instead of returning a list, it returns a NumPy array.

Syntax:

```
numpy.arange(start, stop, step)
```

- **start**: The starting value of the sequence.
- **stop**: The end value (the array will stop just before this number).
- **step**: The step size between numbers (optional, default is 1).

Example:

```
arr = np.arange(1, 10, 2)
print(arr)  # Output: [1 3 5 7 9]
```

Using `numpy.linspace()`

`numpy.linspace()` creates an array of evenly spaced numbers over a specified range. This is useful when you need a fixed number of points between a start and stop value, regardless of the step size.

Syntax:

```
numpy.linspace(start, stop, num)
```

- **start**: The starting value.
- **stop**: The ending value.
- **num**: The number of values to generate (default is 50).

Example:

```
arr = np.linspace(0, 1, 5)
print(arr)  # Output: [0.   0.25 0.5  0.75 1.  ]
```

Using `numpy.zeros()`

`numpy.zeros()` creates an array filled with zeros. You can specify the shape of the array by passing a tuple.

Example:

```
arr = np.zeros((2, 3))  # 2 rows and 3 columns of zeros
print(arr)
```

This creates a 2×3 array filled with zeros:

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

Generating Random Arrays with `numpy.random()`

NumPy has a sub-module called `numpy.random` that allows you to generate arrays filled with random numbers. There are many functions for different kinds of random data:

1. **Random floats between 0 and 1**:

   ```
   random_arr = np.random.rand(2, 3)  # 2×3 array of random floats between 0 and 1
   print(random_arr)
   ```

2. **Random integers**:

   ```
   random_ints = np.random.randint(0, 10, size=(2, 3))  # 2×3 array of random integers between 0 and 10
   print(random_ints)
   ```

3. **Random numbers from a normal distribution**:

```
random_normals = np.random.randn(2, 3)  # 2×3 array of random numbers from the standard normal distribution
print(random_normals)
```

Reshaping and Resizing Arrays

You can change the shape and size of NumPy arrays using `reshape()` and `resize()` methods. This allows you to reorganize the data without changing the data itself.

1. **Reshaping an array**:
   The `reshape()` function lets you change the shape of an array. The total number of elements must stay the same before and after reshaping.

Example:

```
arr = np.arange(6)
reshaped_arr = arr.reshape(2, 3)  # Reshape into 2 rows and 3 columns
print(reshaped_arr)
```

1. **Resizing an array**:
   The `resize()` function changes the shape of an array **in-place**, meaning the array is modified directly. Unlike `reshape()`, you can also resize the array to a larger size (with added zeros if needed).

Example:

```
arr = np.arange(6)
arr.resize((3, 2))  # Resize to 3 rows and 2 columns
print(arr)
```

Recap

- **Creating arrays**: You can create arrays from lists and tuples using `numpy.array()`. For evenly spaced numbers, use `numpy.arange()` and `numpy.linspace()`.
- **Filling arrays**: Use `numpy.zeros()` to create arrays of zeros.
- **Random arrays**: Use `numpy.random` to generate random numbers.
- **Reshaping arrays**: Use `reshape()` to change the shape of an array, and `resize()` to modify it in-place.

These methods help you generate and manipulate arrays, which are fundamental for numerical calculations and data manipulation in Data Science.

III . NumPy Array Attributes

NumPy arrays have several important attributes that help you understand the structure and details of the data they hold. These attributes are easy to use and are essential for working with arrays effectively. Let's explore them!

Understanding Array Dimensions (`ndim` , `shape` , `size`)

1. `ndim`:
   This attribute tells you the number of dimensions an array has. For example:
   - A **1D array** (like a list) has 1 dimension.
   - A **2D array** (like a matrix) has 2 dimensions, and so on.

Example:

```
import numpy as np
arr_1d = np.array([1, 2, 3])
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

print("1D Array Dimensions:", arr_1d.ndim)  # Output: 1
print("2D Array Dimensions:", arr_2d.ndim)  # Output: 2
```

1. `shape`:
   This tells you the size of the array along each dimension (i.e., how many rows and columns it has). For example:
   - A 1D array might have a shape like `(5,)`, which means it has 5 elements.
   - A 2D array might have a shape like `(2, 3)`, which means it has 2 rows and 3 columns.

Example:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Array Shape:", arr.shape)  # Output: (2, 3) - 2 rows, 3 columns
```

1. `size`:
   This tells you the total number of elements in the array (i.e., the product of the dimensions in the `shape`).

Example:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Array Size:", arr.size)  # Output: 6 (2 rows * 3 columns)
```

Array Data Types (`dtype`)

Each element in a NumPy array has a specific data type. The `dtype` attribute allows you to check the data type of the elements in the array. NumPy supports many different types, including integers, floats, booleans, and more.

You can check the `dtype` of an array by accessing this attribute.

Example:

```
arr_int = np.array([1, 2, 3])
arr_float = np.array([1.1, 2.2, 3.3])

print("Data type of arr_int:", arr_int.dtype)  # Output: int64 (or similar based on system)
print("Data type of arr_float:", arr_float.dtype)  # Output: float64 (or similar based on system)
```

You can also specify the data type when creating an array:

```
arr = np.array([1, 2, 3], dtype=float)  # Creating an array with float data type
print(arr.dtype)  # Output: float64
```

Memory Layout of NumPy Arrays

NumPy arrays are stored in memory in a contiguous block, meaning all elements are stored together in one place. This makes accessing elements fast and efficient. Understanding the memory layout is useful when you want to perform operations like reshaping or when working with large datasets.

1. `itemsize`:
   This tells you the number of bytes used to store each element in the array. It depends on the data type. For example, an integer might use 4 bytes, while a float could use 8 bytes.

Example:

```
arr = np.array([1, 2, 3], dtype=int)
print("Item size:", arr.itemsize)  # Output: 8 (on most systems, int64 takes 8 bytes)
```

1. `data`:
   The
   `data` attribute contains the raw bytes of the array. Typically, you won't need to interact directly with this unless you’re dealing with very low-level operations. However, it can be useful for certain optimizations.

Example:

```
arr = np.array([1, 2, 3])
print("Raw data of the array:", arr.data)  # Output: <memory at 0x7fcde7c7c080> (memory locatio
n)
```

Recap

- **Dimensions (`ndim`)**: The number of dimensions (e.g., 1D, 2D, 3D).
- **Shape (`shape`)**: The size of the array in each dimension (rows, columns).

- **Size (`size`)**: The total number of elements in the array.
- **Data type (`dtype`)**: The type of data the array holds (e.g., integers, floats).
- **Item size (`itemsize`)**: The size in bytes of each element.
- **Data (`data`)**: The raw bytes in memory (low-level attribute).

These attributes are useful for understanding the structure of your array and for optimizing memory usage when working with large datasets.

IV . Array Indexing and Slicing

NumPy arrays provide powerful tools for accessing, slicing, and manipulating data. In this section, we'll learn how to access elements in a NumPy array using indexing, extract subsets of data with slicing, and work with multi-dimensional arrays and advanced indexing techniques.

Accessing Array Elements Using Indexing

Indexing in NumPy allows you to access individual elements of an array using their position (or index). Just like lists in Python, NumPy arrays use **0-based indexing** (i.e., the first element is at index 0).

1D Array Indexing:

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])  # Output: 10 (first element)
print(arr[2])  # Output: 30 (third element)
```

Negative Indexing:
You can also use
negative indexing to access elements from the end of the array.

```
print(arr[-1])  # Output: 50 (last element)
print(arr[-2])  # Output: 40 (second to last element)
```

Slicing Arrays to Extract Subsets

Slicing allows you to extract a subset (or "slice") of the array by specifying a range of indices. You can slice arrays in both 1D and multi-dimensional arrays.

1D Array Slicing:
You use the colon (
`:`) operator to slice arrays. The syntax is:

```
array[start:end:step]
```

- **start**: The index where the slice starts (inclusive).

- **end**: The index where the slice ends (exclusive).
- **step**: (optional) The step size between elements.

Example:

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4])  # Output: [20 30 40] (from index 1 to 3)
print(arr[::2])  # Output: [10 30 50] (every 2nd element)
```

Slicing with Negative Indices:
You can also slice using negative indices to count from the end of the array.

```
print(arr[-4:-1])  # Output: [20 30 40] (from index -4 to -2)
```

Multi-Dimensional Array Indexing

In multi-dimensional arrays (like 2D arrays), indexing and slicing become a little more advanced. Each dimension is separated by a comma.

2D Array Indexing:
Here, you specify row and column indices separated by a comma. The syntax is:

```
array[row_index, column_index]
```

Example:

```
arr_2d = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
print(arr_2d[1, 2])  # Output: 60 (element at row 1, column 2)
```

2D Array Slicing:
You can slice rows and columns similarly to how you would in a 1D array.

```
print(arr_2d[0:2, 1:3])  # Output: [[20 30] [50 60]] (first two rows, last two columns)
```

You can also slice specific rows or columns:

```
print(arr_2d[1, :])  # Output: [40 50 60] (second row, all columns)
print(arr_2d[:, 2])  # Output: [30 60 90] (all rows, third column)
```

Advanced Indexing Techniques

1. **Fancy Indexing (Integer Array Indexing)**:
   Fancy indexing allows you to use an array of indices to select specific elements. This is useful when you want to access multiple elements at once, but they are not contiguous.

Example:

```
arr = np.array([10, 20, 30, 40, 50])
indices = [0, 2, 4]  # Select elements at these indices
print(arr[indices])  # Output: [10 30 50]
```

1. **Boolean Indexing**:
   Boolean indexing allows you to select elements based on a condition. You create a boolean array (True or False) based on some condition and use that array to index the original array.

Example:

```
arr = np.array([10, 20, 30, 40, 50])
condition = arr > 25  # Create a boolean array where values are greater than 25
print(arr[condition])  # Output: [30 40 50]
```

You can also combine multiple conditions using logical operators like `&` (AND) or `|` (OR):

```
condition = (arr > 20) & (arr < 50)  # Values greater than 20 but less than 50
print(arr[condition])  # Output: [30 40]
```

Recap

- **Indexing**: Access elements using their index (starting from 0), and use negative indexing to access elements from the end of the array.
- **Slicing**: Extract subsets of arrays by specifying a range of indices using the colon (`:`) operator.
- **Multi-Dimensional Indexing**: For 2D or higher-dimensional arrays, use a comma to separate the indices for rows and columns.
- **Advanced Indexing**:
  - **Fancy Indexing**: Use arrays of indices to select multiple elements at once.
  - **Boolean Indexing**: Use a condition to create a boolean mask and select elements that meet that condition.

These techniques give you powerful control over your NumPy arrays and allow you to easily manipulate and analyze data.

V. Array Operations

In this section, we will explore how to perform various operations on NumPy arrays, including basic arithmetic, element-wise operations, mathematical functions, and universal functions (ufuncs). These operations are essential when working with large datasets and performing calculations in Data Science.

Basic Arithmetic Operations

NumPy arrays allow you to perform arithmetic operations like addition, subtraction, multiplication, and division directly on the arrays. These operations are applied element-wise (i.e., each element in the array is operated on individually).

1. **Addition**:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = arr1 + arr2
print(result)  # Output: [5 7 9]
```

1. **Subtraction**:

```
result = arr1 - arr2
print(result)  # Output: [-3 -3 -3]
```

1. **Multiplication**:

```
result = arr1 * arr2
print(result)  # Output: [4 10 18]
```

1. **Division**:

```
result = arr1 / arr2
print(result)  # Output: [0.25 0.4 0.5]
```

These operations are done **element-wise**, meaning that the operation is applied to each pair of elements from the two arrays.

Element-wise Operations and Broadcasting

Element-wise operations: As we saw in the arithmetic examples above, NumPy supports performing operations element-wise on arrays. This is incredibly useful when working with numerical data, as it allows you to apply calculations without needing loops.

Broadcasting: NumPy allows arrays of different shapes to work together. This is known as **broadcasting**. It automatically adjusts the smaller array to match the shape of the larger array for element-wise operations.

Example:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([10])
result = arr1 + arr2  # Broadcasting happens here
print(result)  # Output: [11 12 13]
```

Here, `arr2` with just one element `[10]` is broadcast to match the shape of `arr1`, so each element in `arr1` is added to `10`.

Mathematical Functions

NumPy provides several useful mathematical functions that help you quickly compute common statistics and metrics on arrays.

1. **Sum (`np.sum()`)**:
   This function calculates the sum of all elements in an array or along a specific axis.

Example:

```
arr = np.array([1, 2, 3, 4])
total = np.sum(arr)
print(total)  # Output: 10
```

1. **Mean (`np.mean()`)**:
   This function calculates the average (mean) of the elements in the array.

Example:

```
mean = np.mean(arr)
print(mean)  # Output: 2.5
```

1. **Standard Deviation (`np.std()`)**:
   This function calculates the standard deviation, which tells you how spread out the numbers are from the mean.

Example:

```
std_dev = np.std(arr)
print(std_dev)  # Output: 1.118 (approximately)
```

These functions are very useful when working with datasets and performing basic statistical analysis.

Universal Functions (ufuncs)

Universal functions (ufuncs) are a core feature of NumPy. They are functions that operate element-wise on arrays. Ufuncs are highly optimized and allow you to apply mathematical operations efficiently to arrays.

Some commonly used ufuncs:

1. **Square Root (`np.sqrt()`)**:
   Calculates the square root of each element in the array.

```
arr = np.array([1, 4, 9, 16])
sqrt_arr = np.sqrt(arr)
```

```
print(sqrt_arr)  # Output: [1. 2. 3. 4.]
```

2. **Exponential (`np.exp()`)**:
   Computes the exponential of each element (e^x).

```
arr = np.array([0, 1, 2])
exp_arr = np.exp(arr)
print(exp_arr)  # Output: [1.        2.71828183 7.3890561 ]
```

3. **Logarithm (`np.log()`)**:
   Computes the natural logarithm (base e) of each element.

```
arr = np.array([1, np.e, np.e**2])
log_arr = np.log(arr)
print(log_arr)  # Output: [0. 1. 2.]
```

4. **Trigonometric Functions**:
   You can use functions like
   `np.sin()`, `np.cos()`, and `np.tan()` for element-wise trigonometric calculations.

```
arr = np.array([0, np.pi/2, np.pi])
sin_arr = np.sin(arr)
print(sin_arr)  # Output: [0. 1. 0.]
```

Recap

- **Basic Arithmetic Operations**: You can perform addition, subtraction, multiplication, and division directly on arrays, and these operations are applied element-wise.
- **Element-wise Operations**: NumPy allows element-wise operations without the need for loops. Broadcasting lets arrays of different shapes interact with each other for element-wise operations.
- **Mathematical Functions**: Functions like `np.sum()`, `np.mean()`, and `np.std()` are helpful for basic statistical analysis.
- **Universal Functions (ufuncs)**: Ufuncs are optimized functions that perform element-wise operations on arrays, like calculating square roots, exponentials, logarithms, and trigonometric functions.

These operations are fundamental for processing and analyzing numerical data efficiently.

6. Reshaping and Concatenating Arrays

In this section, we will learn how to change the shape of arrays and combine multiple arrays into a single array using stacking and concatenating. These operations are essential when you want to organize data in specific formats or combine different datasets.

Changing the Shape of Arrays

1. `reshape()`:
   The `reshape()` function is used to change the shape of an array without changing its data. The total number of elements in the array must stay the same before and after reshaping. You specify the new shape you want by passing a tuple that represents the number of rows and columns (or more dimensions).

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3)  # 2 rows, 3 columns
print(reshaped_arr)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Important: If the total number of elements doesn't match, you will get an error. For example, reshaping a 6-element array into `(3, 3)` will fail because 3 × 3 = 9, which doesn't match the original size.

1. `flatten()`:
   The `flatten()` method is used to convert a multi-dimensional array into a 1D array (a flat array). It returns a copy of the original array.

Example:

```
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
flattened_arr = arr_2d.flatten()
print(flattened_arr)
# Output: [1 2 3 4 5 6]
```

1. `ravel()`:
   Similar to `flatten()`, the `ravel()` function also converts a multi-dimensional array into a 1D array. However, `ravel()` returns a **view** of the original array if possible, instead of a copy (this means changes to the new array might affect the original one).

Example:

```
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
raveled_arr = arr_2d.ravel()
print(raveled_arr)
# Output: [1 2 3 4 5 6]
```

Stacking and Concatenating Arrays

1. **Vertical Stacking (`np.vstack()`)**:

   `np.vstack()` is used to stack arrays vertically (row-wise). The arrays must have the same number of columns.

Example:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
stacked_arr = np.vstack((arr1, arr2))  # Stacks arr1 and arr2 vertically
print(stacked_arr)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

1. **Horizontal Stacking (`np.hstack()`)**:

   `np.hstack()` is used to stack arrays horizontally (column-wise). The arrays must have the same number of rows.

Example:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
stacked_arr = np.hstack((arr1, arr2))  # Stacks arr1 and arr2 horizontally
print(stacked_arr)
# Output: [1 2 3 4 5 6]
```

1. **Concatenating Arrays (`np.concatenate()`)**:

   `np.concatenate()` can be used to join arrays along a specified axis (either vertical or horizontal). This is a more general function than `vstack()` and `hstack()` because it allows you to specify the axis on which to concatenate.

- **Along axis 0 (vertically)**:

  ```
  arr1 = np.array([1, 2, 3])
  arr2 = np.array([4, 5, 6])
  concatenated_arr = np.concatenate((arr1, arr2), axis=0)
  print(concatenated_arr)
  # Output: [1 2 3 4 5 6]
  ```

- **Along axis 1 (horizontally)**:
  For concatenating arrays along columns (horizontally), both arrays must have the same number of rows.

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
concatenated_arr = np.concatenate((arr1, arr2), axis=1)
print(concatenated_arr)
# Output: [[1 2 5 6]
#          [3 4 7 8]]
```

Note: When using `np.concatenate()`, the number of dimensions and the axis you choose must match for the arrays you're concatenating.

Recap

- **Reshaping**:
  - `reshape()`: Change the shape of the array (dimensions).
  - `flatten()`: Convert a multi-dimensional array into a 1D array.
  - `ravel()`: Similar to `flatten()`, but returns a view instead of a copy.
- **Stacking Arrays**:
  - `np.vstack()`: Stack arrays vertically (row-wise).
  - `np.hstack()`: Stack arrays horizontally (column-wise).
- **Concatenating Arrays**:
  - `np.concatenate()`: Concatenate arrays along a specified axis (axis=0 for vertical, axis=1 for horizontal).

These operations help in restructuring and combining arrays efficiently when dealing with large datasets.