1. Getting familiar with Numpy:

Intializing of an numpy:

```
import numpy as np
```

Creation of Array : here i created a one dimensional array.

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

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

Creation of 2D Array:

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

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

NumPy’s core functionalities: functionalities like

- **np.zeros((3, 3)):** Creates a 3x3 array filled with zeros.
- **np.ones((2, 4)):** Creates a 2x4 array filled with ones.
- **np.arange(0, 10, 2):** Creates an array with values from 0 to 10 with a step of 2.

```
zeros_array = np.zeros((3, 3))
ones_array = np.ones((2, 4))
range_array = np.arange(0, 10, 2)
```

Array Properties:

- **Shape**: Returns the dimensions of the array.
- **Size**: Returns the total number of elements.
- **Data Type**: Returns the data type of the array elements.

```
print(arr.shape)
print(arr.size)
print(arr.dtype)
```

```
(2, 3)
6
int64
```

Basic operations:

Arithmetic Operations:

- Addition
- Subtraction
- Multiplication
- Division

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

sum_array = array1 + array2  # addition
diff_array = array1 - array2  # subtraction
prod_array = array1 * array2  # multiplication
div_array = array1 / array2  # division

print(sum_array)
print(diff_array)
print(prod_array)
print(div_array)
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
```

2. Data Manipulation:

Demonstrating data manipulation using Numpy, focusing on array creation, indexing, slicing, reshaping, and applying mathematical operations.

Array Creation :

```
data = np.array([10, 20, 30, 40, 50, 60])
```

Indexing and Slicing:

- Indexing used to specify the element.
- Slicing used to access the specific range of elements by mentioning their indices.

```
print("Original Array:", data)
print("Element at index 2:", data[2])
print("Slicing from index 1 to 4:", data[1:5])
```

```
Original Array: [10 20 30 40 50 60]
Element at index 2: 30
Slicing from index 1 to 4: [20 30 40 50]
```

Reshaping: Used to change the shape.

```
reshaped_data = data.reshape(2, 3)
print("Reshaped Array:\n", reshaped_data)
```

```
Reshaped Array:
 [[10 20 30]
 [40 50 60]]
```

Mathematical Operations:

```
doubled_data = data * 2
print("Doubled Array:", doubled_data)
```

```
Doubled Array: [ 20  40  60  80 100 120]
```

3. Data Aggregation:

Computing summary statistics like mean, median, standard deviation, and sum using Numpy. Practice grouping data and performing aggregations.

Summary Statistics: statistics contains mean , median , standard deviation.

```
mean_value = np.mean(data)
median_value = np.median(data)
std_deviation = np.std(data)
total_sum = np.sum(data)



print("Mean:", mean_value)
print("Median:", median_value)
print("Standard Deviation:", std_deviation)
print("Sum:", total_sum)
```

```
Mean: 35.0
Median: 35.0
Standard Deviation: 17.07825127659933
Sum: 210
```

Grouping Data Example: By the grouped data here i perform the sum across the rows and columns.

```
grouped_data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
row_sums = np.sum(grouped_data, axis=1)  # Sum across rows
col_sums = np.sum(grouped_data, axis=0)  # Sum across columns

print("Row Sums:", row_sums)
print("Column Sums:", col_sums)
```

```
Row Sums: [ 6 15 24]
Column Sums: [12 15 18]
```

4. Data Analysis:

Performing data analysis using Numpy to find correlations, identify outliers, and calculate percentiles. Highlight Numpy’s efficiency in handling large datasets.

```
data1 = np.array([100, 200, 300, 400, 500])
data2 = np.array([10, 20, 30, 40, 50])
```

Correlation:

```
correlation = np.corrcoef(data1, data2)[0, 1]
print("Correlation between data1 and data2:", correlation)
```

```
Correlation between data1 and data2: 0.9999999999999999
```

Identifying Outliers:

```
outliers = data1[data1 > 400]
print("Outliers:", outliers)
```

```
Outliers: [500]
```

Calculating Percentiles:

```
percentile_50 = np.percentile(data1, 50)
percentile_90 = np.percentile(data1, 90)

print("50th Percentile:", percentile_50)
print("90th Percentile:", percentile_90)
```

```
50th Percentile: 300.0
90th Percentile: 460.0
```

5. Application in Data Science:

Certainly! Here's a detailed conclusion on how NumPy is beneficial for data science professionals, and its advantages over traditional Python data structures, along with real-world applications:

Conclusion on the Use of NumPy in Data Science:

Benefits of NumPy for Data Science Professionals

NumPy, short for Numerical Python, is a fundamental package for scientific computing in Python. It offers a powerful and flexible way to perform numerical operations efficiently, making it an indispensable tool for data science professionals. Here's why NumPy is so valuable:

Performance Efficiency: NumPy's core data structure, the ndarray (n-dimensional array), allows for highly efficient storage and manipulation of large numerical datasets. Unlike traditional Python lists or arrays, which are slower and less memory-efficient for large-scale operations, NumPy arrays are implemented in C and are optimized for performance. This leads to significant speed improvements in computation-heavy tasks.

Vectorization: NumPy supports vectorized operations, which means that operations are applied element-wise on arrays without the need for explicit loops. This vectorization can reduce the amount of code and speed up execution by leveraging low-level optimizations. For example, adding two large arrays element-wise with NumPy is much faster than using a loop in standard Python.

Broad Range of Mathematical Functions: NumPy provides a wide array of mathematical functions and operations, including linear algebra, statistical analysis, Fourier transforms, and random number generation. These functions are highly optimized and can be used seamlessly with NumPy arrays.

Interoperability: NumPy arrays are compatible with many other scientific libraries, including SciPy, pandas, and scikit-learn. This interoperability makes it easier to build comprehensive data science workflows, as NumPy serves as a foundational element in the ecosystem of scientific computing libraries.

Ease of Use: Despite its power, NumPy is relatively easy to learn and use. Its array operations are straightforward and intuitive, and the library comes with extensive documentation and community support.

Advantages Over Traditional Python Data Structures

Traditional Python data structures such as lists, tuples, and dictionaries are versatile and useful for many tasks. However, they fall short when it comes to numerical computations:

Performance: Python lists are slower for numerical operations because they do not support efficient element-wise operations or memory management. Each element in a Python list can be of different types, which introduces overhead in numerical computations.

Memory Efficiency: NumPy arrays use contiguous blocks of memory and have a fixed type for all elements, which allows them to be more memory-efficient than Python lists, where each element is an object with its own metadata.

Functionality: NumPy offers specialized functions for mathematical operations that are not available with standard Python data structures. For example, NumPy’s linalg module provides advanced linear algebra operations, which would require custom implementation or third-party libraries with Python lists.

Real-World Examples Where NumPy’s Capabilities Are Crucial

Machine Learning: In machine learning, NumPy is often used for preprocessing data, performing matrix operations, and implementing algorithms. For instance, libraries like scikit-learn and TensorFlow rely on NumPy arrays for their internal computations. NumPy's efficiency in handling large datasets and performing mathematical operations is critical for training and evaluating machine learning models.

Financial Analysis: In financial analysis, NumPy helps with the manipulation of time-series data, calculation of financial indicators, and performing quantitative analyses. For example, NumPy can be used to calculate moving averages, volatility, and other statistical metrics crucial for investment decision-making and risk assessment.

Scientific Research: NumPy is widely used in scientific research for tasks such as numerical simulations, data fitting, and statistical analysis. In fields like physics, biology, and engineering, researchers use NumPy to handle large datasets and perform complex calculations efficiently. For instance, simulations of physical systems or the analysis of experimental data often rely on NumPy’s mathematical functions and array operations.