01 - Introduction to Numpy

NumPy (Numerical Python) is a powerful library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. There are several reasons why you should consider using NumPy in your Python projects:

1. NumPy provides fast and efficient operations on large arrays and matrices.
2. It offers a wide range of mathematical functions for numerical computations.
3. NumPy's multi-dimensional arrays enable efficient storage and manipulation of data.
4. Broadcasting allows for efficient operations on arrays with different shapes.
5. NumPy integrates seamlessly with other scientific libraries, expanding its capabilities.
6. It provides advanced indexing and slicing options for data extraction and manipulation.
7. NumPy's optimized C code delivers high-performance computing compared to Python lists.
8. The library offers tools for file input/output and data persistence.
9. NumPy is widely used and has extensive community support, ensuring its reliability and resources.
10. It serves as a foundation for many scientific and data analysis tasks in Python.

Don't worry if these reasons don't make enough sense to you at this point. You will learn all of these features very soon in the upcoming sections.

01.1 - How to use 'Numpy'?

To use NumPy in your Python code, you need to follow these steps:

1. **Install NumPy:** If you haven't already installed NumPy, you can use pip, the Python package installer, by running the command `pip install numpy` in your terminal or command prompt.

2. **Import NumPy:** Once NumPy is installed, you need to import it into your Python script using the import statement as shown below.

In [1]:
```python
import numpy as np
```

01.2 - The Numpy Arrays

The soul of Numpy is the 'Numpy Arrays'. In NumPy, an array is like a container that can hold a collection of values. It's similar to a list in Python, but with some additional benefits. Arrays can be one-dimensional (1D), two-dimensional (2D), or even three-dimensional (3D).

1. **1D Arrays** can be thought of as a single row of values. For example, imagine you have a list of temperatures recorded each day for a week: [25, 26, 24, 28, 27, 25, 23]. In NumPy, you can create a 1D array to store these values and perform operations on them more efficiently.

2. **2D Arrays** are like a grid or a table. You can think of them as having rows and columns, just like a spreadsheet. For instance, consider a list of students and their corresponding marks in three subjects: Math, English, and Science. You can represent this data in a 2D array, where each row represents a student and each column represents a subject. This way, you can organize and analyze the data effectively.

3. **3D Arrays** add another dimension to the grid. They can be visualized as a stack of 2D arrays. For example, imagine you have a collection of grayscale images, where each image is a 2D array representing pixel intensities. If you want to store multiple images, you can use a 3D array. The extra dimension allows you to stack multiple 2D images together, making it easier to process and manipulate the entire collection.

Now, let's discuss the concept of **axes** in NumPy arrays. Each dimension of an array is called an axis. In a 2D array, the first axis refers to the rows, and the second axis refers to the columns.

To illustrate this, imagine you have a 2D array representing a grid of numbers, where rows represent different stores, and columns represent the sales of different products. If you want to calculate the total sales for each store (across columns), you would sum along the axis 1 (columns). On the other hand, if you want to calculate the total sales for each product (across rows), you would sum along the axis 0 (rows).

In [2]:

```python
### 1D Array Example
temperatures = [25, 26, 24, 28, 27, 25, 23]
temperatures_array = np.array(temperatures)
print(temperatures_array)

### 2D Array Example
marks = [
    [80, 75, 85],  # Marks of the first student [Math, English, Science]
    [90, 88, 92],  # Marks of the second student [Math, English, Science]
    [77, 80, 85],  # Marks of the third student [Math, English, Science]
]
marks_array = np.array(marks)
print(marks_array)

### 3D Array Example
image1 = [
    [100, 120, 105],
    [115, 110, 90],
    [90, 95, 100]
]
image2 = [
    [80, 70, 85],
    [95, 100, 110],
    [105, 100, 95]
]
images = [image1, image2]
images_array = np.array(images)
print(images_array)
```

```
[25 26 24 28 27 25 23]
[[80 75 85]
 [90 88 92]
 [77 80 85]]
[[[100 120 105]
  [115 110  90]
  [ 90  95 100]]

 [[ 80  70  85]
  [ 95 100 110]
  [105 100  95]]]
```

Let's explore the functions in NumPy that allow us to create arrays with initial placeholders.

1. **np.zeros()**: This function creates an array filled with zeros.
   Example: Suppose you want to create an array of size 5, representing the number of seats available in a row in a movie theater. You can use np.zeros() to create the array with all seats initially set to zero.

2. **np.ones():** This function creates an array filled with ones.
   Example: Let's say you want to create an array to represent the number of goals scored by a football team in five matches. You can use np.ones() to create the array with all elements initially set to one.

3. **np.arange():** This function creates an array with a sequence of numbers.
   Example: Imagine you need an array containing numbers from 0 to 9. You can utilize np.arange() to create the array with the desired sequence.

4. **np.linspace():** This function creates an array with evenly spaced values between a specified range.
   Example: Suppose you want to create an array of 6 values ranging from 0 to 1, representing intervals of time. You can use np.linspace() to achieve this.

5. **np.full():** This function creates an array with all elements set to a specific value.
   Example: Let's say you want to create an array representing the brightness levels of pixels in a grayscale image, where all pixels have an initial brightness value of 128. You can utilize np.full() to create the array with the desired value.

6. **np.empty():** This function creates an array without initializing its elements to any particular value. The values in the array can be arbitrary and vary each time the function is called.
   Example: Suppose you need an array of size 4, representing the scores of

participants in a competition. You can use np.empty() to create the array without initializing the values.

```
In [3]: ### Examples

        seats = np.zeros(5)
        print(seats)

        goals = np.ones(5)
        print(goals)

        numbers = np.arange(10)
        print(numbers)

        time_intervals = np.linspace(0, 1, 6)
        print(time_intervals)

        brightness_levels = np.full((3, 3), 128)
        print(brightness_levels)

        scores = np.empty(4)
        print(scores)
```

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
[0 1 2 3 4 5 6 7 8 9]
[0.  0.2 0.4 0.6 0.8 1. ]
[[128 128 128]
 [128 128 128]
 [128 128 128]]
[0.00000000e+000 2.26193921e-314 2.26196169e-314 0.00000000e+000]
```

The values in the output of np.empty() are not predictable and can vary each time the function is called. This is because np.empty() does not initialize the array elements to any particular value. Instead, it allocates memory for the array without setting the values explicitly.

You might wonder why one would use np.empty() instead of np.zeros() or np.ones(). The reason is performance. np.empty() is faster than np.zeros() or np.ones() because it does not waste time initializing the values. It can be useful when you know you will be overwriting the array values with your own data shortly after its creation.

Example: Suppose you have a simulation where you need to generate a large array and then fill it with calculated values. You can use np.empty() to create the array efficiently without wasting time initializing the elements.

```
import numpy as np

size = 1000000
simulation_data = np.empty(size)
# Perform calculations to fill the array with meaningful values
# ...
```

In the example above, we create an array of size 1,000,000 using np.empty(). We then perform calculations to fill the array with meaningful values specific to our simulation. Since we will overwrite the values anyway, using np.empty() helps save unnecessary initialization time compared to np.zeros() or np.ones().

It's important to note that while np.empty() can be more efficient, it leaves the array with uninitialized values, and accessing those values without proper assignment may lead to unexpected results. Therefore, it is essential to ensure you initialize or overwrite the values in the array appropriately after using np.empty().

In summary, the functions np.zeros(), np.ones(), np.arange(), np.linspace(), np.full(), and np.empty() provide convenient ways to create NumPy arrays with initial placeholders.

01.3 - I/O with Numpy Arrays

In NumPy, I/O refers to input/output operations that allow us to read data from external files or write data to files. NumPy provides functions for efficient I/O operations, making it easier to work with data in various formats. Let's explore why and when we might need to use NumPy I/O functions, and introduce some common ones.

Why should we use NumPy I/O?

- NumPy I/O functions provide efficient and convenient ways to read and write data in various formats, such as text files, binary files, and NumPy's own .npy format.
- NumPy I/O functions are optimized for working with large datasets, allowing for faster processing and reduced memory consumption.
- NumPy I/O functions seamlessly integrate with other NumPy functions, enabling smooth data manipulation and analysis workflows.

When might we need to use NumPy I/O?

- When we have large datasets that are stored in external files and we want to read them into NumPy arrays for further processing or analysis.
- When we want to save NumPy arrays to files, either for later use or for sharing with others.
- When we need to exchange data with other scientific computing libraries or tools that support NumPy-compatible file formats.

Now, let's introduce some commonly used NumPy I/O functions along with some examples.

1. **np.load()** and **np.save()**:

- np.load() is used to load data from a .npy file into a NumPy array.
- np.save() is used to save a NumPy array to a .npy file.

2. **np.loadtxt()** and **np.savetxt()**:

- np.loadtxt() is used to load data from a text file into a NumPy array.

- np.savetxt() is used to save a NumPy array to a text file.

In [4]:
```python
student_names = np.array(['Alice', 'Bob', 'Charlie', 'David'])
np.save('student_names.npy', student_names)

loaded_names = np.load('student_names.npy')
print(loaded_names)
```

```
['Alice' 'Bob' 'Charlie' 'David']
```

In this example, we use np.save() to save the student_names array to a file. Later, we use np.load() to load the saved array from the file and assign it to the loaded_names array.

Suppose you have a text file named 'data.txt' containing comma-separated values (CSV) representing student scores:

```
Alice,95
Bob,87
Charlie,92
David,88
```

In [5]:
```python
data = np.loadtxt('data.txt', delimiter=',', dtype=str)
print(data)

scores = np.array([95, 87, 92, 88])
np.savetxt('scores.txt', scores, delimiter=',')
```

```
[['Alice' '95']
 ['Bob' '87']
 ['Charlie' '92']
 ['David' '88']]
```

In this example, we use np.loadtxt() to load the data from the text file 'data.txt'. We specify the delimiter as , to indicate that the values are comma-separated. The resulting NumPy array, data, contains the loaded data.

We then save the scores array to a text file named 'scores.txt', with values separated by commas.

In summary, NumPy provides functions like `np.load()`, `np.save()`, `np.loadtxt()`, and `np.savetxt()` to facilitate input/output operations with external files. These functions allow us to efficiently read data from files into NumPy arrays or save NumPy arrays to files, enabling seamless integration with various data formats and external tools. Whether it's loading pre-existing data or saving results for future use, NumPy I/O functions offer convenient ways to work with data in real-life scenarios.

01.4 - Inspecting your Numpy Array

When working with NumPy arrays, it's essential to inspect their properties to understand their dimensions, shape, size, and data type. Let's explore some key attributes for inspecting NumPy arrays:

1. **a.shape:**

- This attribute returns a tuple indicating the dimensions of the array.
- The shape attribute provides the size of each dimension, indicating the number of elements along each axis.

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

```
(2, 3)
```

In this example, the NumPy array a has a shape of (2, 3), indicating that it has 2 rows and 3 columns.

2. **a.ndim:**

- This attribute returns an integer indicating the number of dimensions (axes) of the array.

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

```
2
```

In this example, the NumPy array a has a dimension of 2 since it is a 2-dimensional array.

3. **a.size**

- This attribute returns the total number of elements in the array.

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

```
6
```

In this example, the NumPy array a has a total of 6 elements.

4. **a.dtype:**

- This attribute returns the data type of the elements in the array.

In [9]:
```
a = np.array([1, 2, 3])
print(a.dtype)
```

```
int64
```

In this example, the NumPy array a contains integers, and its data type is int64.

Inspecting these attributes provides crucial information about the structure and content of the NumPy array. It allows us to understand the dimensions, shape, size, and data type, which are essential for performing operations, manipulating data, and ensuring compatibility with other array-based functions or libraries.
