Chapter 1

Introduction to NumPy

Welcome to the wonderful world of NumPy! If you're just starting your Python journey and using Google Colab, you're in the right place. This book will guide you through the fundamentals of NumPy, a powerful library that allows you to work with arrays efficiently, paving the way for data analysis, scientific computing, and machine learning.

1.1 Why NumPy? An Intuitive Example

Imagine you're a teacher and you have the scores of your students in a math exam. You want to calculate the average score, find the highest and lowest scores, and perhaps even adjust all the scores by adding a bonus mark. You *could* do this using standard Python lists, but it would involve writing loops and performing calculations one element at a time.

NumPy lets you do all of that, and much more, with concise and efficient code. Think of it like this: NumPy provides you with super-powered spreadsheets (arrays) that Python can understand natively and manipulate with incredible speed.

Let's say you have the following scores: 75, 82, 68, 90, 88.

With NumPy, you can perform operations on this entire set of scores at once, without writing a single loop! This is the magic of **vectorized operations**, a core concept we'll explore in detail.

1.2 What is NumPy?

At its heart, NumPy (short for Numerical Python) provides a powerful object called the **ndarray**, or n-dimensional array. Think of it as a grid of values, all of the same type (usually numbers), organized in a specific number of dimensions. These arrays are not only efficient to store, but they also allow for incredibly fast calculations compared to standard Python lists.

Chapter 2

NumPy Arrays: The Foundation

2.1 Creating NumPy Arrays

The first step is to import NumPy:

```
import numpy as np
```

np is the standard alias for NumPy. Now, let's create some arrays!

- **From a Python List:** The easiest way to create a NumPy array is to convert a Python list:

```
my_list = [1, 2, 3, 4, 5]
my_array = np.array(my_list)
print(my_array)  # Output: [1 2 3 4 5]
```

- **Using Built-in Functions:** NumPy provides several convenient functions to create arrays:

  - `np.zeros(shape)`: Creates an array filled with zeros. `shape` is a tuple defining the dimensions (e.g., `(3, 4)` for a 3x4 array).

```
zeros_array = np.zeros((2, 3))  # 2 rows, 3 columns
print(zeros_array)
# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]]
```

  - `np.ones(shape)`: Creates an array filled with ones.

```
ones_array = np.ones((3, 2))  # 3 rows, 2 columns
print(ones_array)
# Output:
# [[1. 1.]
#  [1. 1.]
#  [1. 1.]]
```

  - `np.arange(start, stop, step)`: Creates an array with values in a given range. Similar to Python's `range()` function, but returns a NumPy array.

```
range_array = np.arange(0, 10, 2)  # Start at 0, stop before
    10, step by 2
print(range_array)  # Output: [0 2 4 6 8]
```

- `np.linspace(start, stop, num)`: Creates an array with evenly spaced values over a specified interval. `num` is the number of elements you want.

```
linspace_array = np.linspace(0, 1, 5)  # 5 evenly spaced
   values between 0 and 1
print(linspace_array)  # Output: [0.   0.25 0.5  0.75 1.  ]
```

2.2 Array Attributes

Before we dive into manipulating arrays, let's understand some key attributes:

- `ndim`: The number of dimensions (axes) of the array.
- `shape`: A tuple indicating the size of each dimension.
- `size`: The total number of elements in the array.
- `dtype`: The data type of the elements in the array (e.g., `int64`, `float64`).

```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
print("Number of dimensions:", my_array.ndim)   # Output: 2
print("Shape:", my_array.shape)           # Output: (2, 3)
print("Size:", my_array.size)             # Output: 6
print("Data type:", my_array.dtype)        # Output: int64 (or
   similar)
```

Chapter 3

Indexing and Slicing

Accessing elements in NumPy arrays is similar to Python lists, but with added flexibility for multi-dimensional arrays.

3.1 Basic Indexing

- **For a 1D array:** `array[index]`

```
my_array = np.array([10, 20, 30, 40, 50])
print(my_array[0])  # Output: 10
print(my_array[3])  # Output: 40
```

- **For a 2D array:** `array[row`$_{i}ndex, column_{i}ndex$`]`

```
my_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(my_array[0, 0])  # Output: 1 (element at row 0, column 0)
print(my_array[1, 2])  # Output: 6 (element at row 1, column 2)
```

3.2 Slicing

Slicing allows you to extract a portion of an array.

- **For a 1D array:** `array[start:stop:step]`

```
my_array = np.array([10, 20, 30, 40, 50])
print(my_array[1:4])    # Output: [20 30 40]
print(my_array[:3])     # Output: [10 20 30]
print(my_array[::2])    # Output: [10 30 50] (every other
   element)
```

- **For a 2D array:** `array[row`$_{s}tart : row_{s}top : row_{s}step, colcol_{s}tart : col_{s}top : col_{s}tep$`]`

```
my_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(my_array[:2, 1:])  # Output: [[2 3] [5 6]] (first two rows
   , columns from index 1 onwards)
```

Chapter 4

Vectorized Operations: The Power of NumPy

This is where NumPy truly shines. Vectorized operations allow you to perform operations on entire arrays (or sections of arrays) *at once*, without explicit loops. This leads to much faster and more concise code.

4.1 Arithmetic Operations

You can perform element-wise arithmetic operations directly on arrays:

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

print(array1 + array2)  # Output: [5 7 9]
print(array1 * 2)      # Output: [2 4 6]
print(array2 / array1)  # Output: [4.  2.5 2. ]
```

4.2 Universal Functions (ufuncs)

NumPy provides a wide range of universal functions (ufuncs) that operate element-wise on arrays. Examples include `np.sin()`, `np.cos()`, `np.exp()`, `np.log()`, and many more.

```
my_array = np.array([0, 1, 2, 3])
print(np.sin(my_array))  # Output: [ 0.          0.84147098
   0.90929743  0.14112001]
```

4.3 Returning to Our Teacher Example

Remember the teacher with the student scores? Let's see how NumPy can simplify their task:

```
import numpy as np

scores = np.array([75, 82, 68, 90, 88])

average_score = np.mean(scores)
highest_score = np.max(scores)
lowest_score = np.min(scores)
bonus_scores = scores + 5  # Add a bonus of 5 to each score

print("Average score:", average_score)
print("Highest score:", highest_score)
print("Lowest score:", lowest_score)
print("Bonus scores:", bonus_scores)
```

This is much cleaner and more efficient than using loops to perform these calculations.

Chapter 5

Conclusion

NumPy arrays, indexing, slicing, and vectorized operations are fundamental concepts for anyone working with numerical data in Python. By mastering these basics, you'll be well-equipped to tackle more advanced data analysis, scientific computing, and machine learning tasks. Experiment with the examples in Google Colab, and don't be afraid to explore the extensive NumPy documentation to discover even more powerful features! Good luck on your NumPy journey!