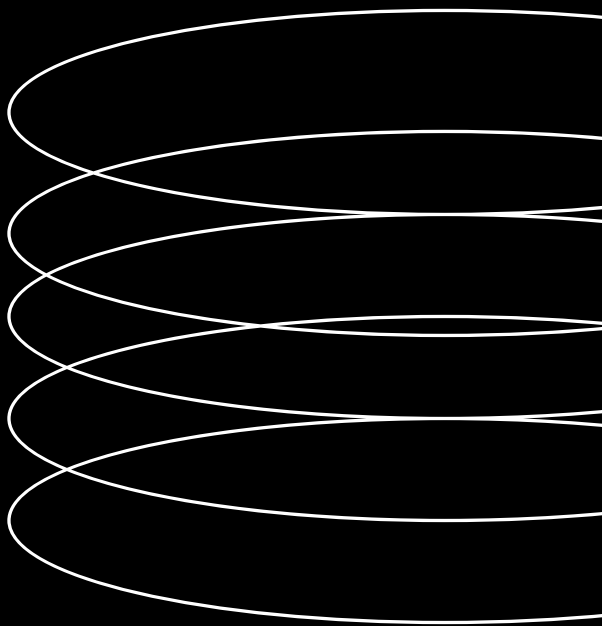
@RAMCHANDRAPADWAL

Practical Guide to NumPy for Data Science



A STEP-BY-STEP GUIDE

Table of Contents

- Introduction to NumPy
  - What is NumPy?
  - Why Use NumPy in Data Science?
- Installing and Importing NumPy
  - 2.1 Installation Instructions
  - 2.2 Importing NumPy
- NumPy Arrays
  - 3.1 Creating NumPy Arrays
  - 3.2 Array Attributes
  - 3.3 Indexing and Slicing Arrays
  - 3.4 Array Operations 3.5 Broadcasting
- Array Manipulation
  - 4.1 Reshaping Arrays
  - 4.2 Joining and Splitting Arrays
  - 4.3 Changing Array Data Types
  - 4.4 Sorting and Searching Arrays
- Mathematical Functions
  - 5.1 Basic Mathematical Operations
  - 5.2 Trigonometric Functions
  - 5.3 Exponential and Logarithmic Functions
  - 5.4 Statistical Functions
  - 5.5 Linear Algebra Operations
- Array Input and Output
  - 6.1 Saving and Loading Arrays
  - 6.2 Text File Input and Output
  - 6.3 Binary File Input and Output
  - 6.4 Compressed File Input and Output
- Advanced NumPy Features
  - 7.1 Broadcasting in Depth
  - 7.2 Array Manipulation Tricks
  - 7.3 Fancy Indexing and Boolean Masking
  - 7.4 Structured Arrays
  - 7.5 Universal Functions
- Performance Tips and Best Practices
  - 8.1 Vectorization
  - 8.2 Memory Efficiency
  - 8.3 Use NumPy Functions Instead of Loops
- Conclusion

@RAMCHANDRAPADWAL

CHAPTER N.1

Introduction to NumPy

A Step-by-Step Guide

1.1 WHAT IS NUMPY?

NumPy (Numerical Python) is an open-source Python library that provides powerful tools for performing numerical computations and working with multi-dimensional arrays. It serves as the foundation for many other scientific computing libraries in Python and is widely used in data science, machine learning, and scientific research.

1.2 WHY USE NUMPY IN DATA SCIENCE?

NumPy offers several advantages for data science tasks:

- Efficient array operations: NumPy provides fast and efficient implementations of mathematical operations on arrays, making it suitable for large-scale data processing.
- Multi-dimensional arrays: NumPy arrays allow for efficient storage and manipulation of multi-dimensional data, such as matrices or tensors.
- Broadcasting: NumPy supports broadcasting, which enables operations between arrays of different shapes, making code concise and readable.
- Integration with other libraries: NumPy seamlessly integrates with other data science libraries such as Pandas, Matplotlib, and scikit-learn, providing a comprehensive ecosystem for data analysis and machine learning.

@RAMCHANDRAPADWAL

CHAPTER N.2

Installing and Importing NumPy

A Step-by-Step Guide

2.1 Installation Instructions

To install NumPy, you can use the Python package manager, pip. Open your command prompt or terminal and run the following command:

```
pip install numpy
```

This will download and install the latest version of NumPy.

2.2 Importing NumPy

Once NumPy is installed, you can import it into your Python script or Jupyter Notebook using the following import statement:

```
import numpy as np
```

The "np" alias is commonly used for NumPy to shorten the code.

@RAMCHANDRAPADWAL

CHAPTER N.3

NumPy Arrays

A Step-by-Step Guide

3.1 Creating NumPy Arrays

1. NumPy arrays are the fundamental data structure in NumPy. They can be created in several ways:

- Using the **numpy.array()** function:

This function takes a sequence-like object (e.g., list, tuple) and creates a NumPy array from it.

```
import numpy as np

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

OUTPUT:

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

- Using the **numpy.arange()** function:

This function creates an array with regularly spaced values within a specified range.

```
arr = np.arange(0, 10, 2)
print(arr)
```

OUTPUT:

```
[0 2 4 6 8]
```

- Using the **numpy.zeros()** and **numpy.ones()** functions:

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

```
zeros = np.zeros((3, 3))
print(zeros)
```

OUTPUT:

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

```
ones = np.ones((2, 4))
print(ones)
```

OUTPUT:

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

3.2 Array Attributes

NumPy arrays have various attributes that provide useful information about the array's shape, size, and data type. Some commonly used attributes include:

- **shape**: Returns a tuple representing the size of each dimension of the array.
- **dtype**: Returns the data type of the array elements.
- **ndim**: Returns the number of array dimensions.
- **size**: Returns the total number of elements in the array.
- **itemsize**: Returns the size in bytes of each array element.

3.3 Indexing and Slicing Arrays

NumPy arrays can be accessed and sliced using indexing and slicing operations. The indexing starts at 0, and negative indices can be used to access elements from the end of the array.

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

print(arr[0])       # Output: 1
print(arr[-1])      # Output: 5
print(arr[2:4])     # Output: [3 4]
```

3.4 Array Operations

NumPy provides a wide range of mathematical and logical operations on arrays. These operations are applied element-wise, and they can be performed between arrays of compatible shapes.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

print(arr1 + arr2)      # Output: [5 7 9]
print(arr1 * arr2)      # Output: [4 10 18]
print(arr1 ** 2)        # Output: [1 4 9]
```

3.5 Broadcasting

Broadcasting is a powerful feature of NumPy that allows arithmetic operations between arrays of different shapes. It eliminates the need for explicit looping and simplifies the code.

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 2

print(arr1 * scalar)
```

OUTPUT:

```
[[ 2  4  6]
 [ 8 10 12]]
```

@RAMCHANDRAPADWAL

CHAPTER N.4

Array Manipulation

A Step-by-Step Guide

4.1 Reshaping Arrays

NumPy provides several methods to reshape arrays, including **reshape()**, **flatten()**, and **ravel()**. Reshaping allows you to change the dimensions of an array without changing the underlying data.

```
arr = np.arange(1, 10)

reshaped = arr.reshape(3, 3)
print(reshaped)
```

OUTPUT:

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

4.2 Joining and Splitting Arrays

NumPy allows you to concatenate or stack arrays together using functions like **concatenate()**, **vstack()**, and **hstack()**. Similarly, you can split arrays into multiple smaller arrays using **split()**, **vsplit()**, and **hsplit()**.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

concatenated = np.concatenate((arr1, arr2))
print(concatenated)
```

OUTPUT:

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

4.3 Changing Array Data Types

NumPy provides functions to change the data type of an array, such as **astype()**. This is particularly useful when you need to convert an array to a different data type for computation or storage.

```
arr = np.array([1, 2, 3], dtype=np.float64)

new_arr = arr.astype(np.int32)
print(new_arr)
```

OUTPUT:

```
[1 2 3]
```

4.4 Sorting and Searching Arrays

NumPy offers functions for sorting arrays, such as **sort()** and **argsort()**. You can also search for elements using functions like **where()** and **searchsorted()**.

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

sorted_arr = np.sort(arr)
print(sorted_arr)
```

OUTPUT:

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

@RAMCHANDRAPADWAL

CHAPTER N.5

Mathematical Functions

A Step-by-Step Guide

5.1 Basic Mathematical Operations

NumPy provides a wide range of mathematical functions for performing basic operations on arrays. These functions include **add()**, **subtract()**, **multiply()**, **divide()**,and more.

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

print(np.square(arr))          # Output: [1 4 9]
print(np.sqrt(arr))            # Output: [1.         1.41421356 1.73205081]
print(np.exp(arr))             # Output: [ 2.71828183  7.3890561  20.08553692]
```

5.2 Trigonometric Functions

NumPy provides various trigonometric functions, such as **sin()**, **cos()**, **tan()**, **arcsin()**, **arccos()**, and **arctan()**. These functions operate element-wise on arrays.

```
arr = np.array([0, np.pi / 2, np.pi])

print(np.sin(arr)) Output: [0.00000000e+00 1.00000000e+00 1.22464680e-16]
print(np.cos(arr)) Output: [1.0000000e+00 6.1232340e-17 -1.0000000e+00]
print(np.tan(arr)) Output: [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

5.3 Exponential and Logarithmic Functions

NumPy provides functions for exponential and logarithmic operations, such as **exp()**, **log()**, **log10()**, **log2()**, **expm1()**, **log1p()**, and more.

```
arr = np.array([1, 10, 100])

print(np.exp(arr))      # Output: [2.71828183e+00 2.20264658e+04 2.68811714e+43]
print(np.log(arr))      # Output: [0.         2.30258509 4.60517019]
print(np.log10(arr))    # Output: [0. 1. 2.]
```

5.4 Statistical Functions

NumPy offers a variety of statistical functions for analyzing arrays, such as **mean()**, **median()**, **std()**, **var()**, **min()**, and **max()**.

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

print(np.mean(arr))             # Output: 3.0
print(np.median(arr))           # Output: 3.0
print(np.std(arr))              # Output: 1.41421356
```

5.5 Linear Algebra Operations

NumPy provides linear algebra functions for matrix operations, such as **dot()**, **transpose()**, **inverse()**, and **eig()**.

```
matrix = np.array([[1, 2], [3, 4]])

print(np.dot(matrix, matrix))   # Output: [[ 7 10]
                                #          [15 22]]
print(np.transpose(matrix))     # Output: [[1 3]
                                #          [2 4]]
```

@RAMCHANDRAPADWAL

CHAPTER N.6

Array Input and Output

A Step-by-Step Guide

6.1 Saving and Loading Arrays

NumPy provides functions to save and load arrays from disk. You can use **np.save()** to save an array to a binary file and **np.load()** to load the array back into memory.

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

np.save('array.npy', arr)
loaded_arr = np.load('array.npy')

print(loaded_arr)
```

OUTPUT:

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

6.2 Text File Input and Output

NumPy allows you to save and load arrays in text file format using **np.savetxt()** and **np.loadtxt()**. This is useful for sharing data with other applications or for human-readable storage.

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

np.savetxt('array.txt', arr)
loaded_arr = np.loadtxt('array.txt')

print(loaded_arr)
```

OUTPUT:

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

6.3 Binary File Input and Output

NumPy supports saving and loading arrays in binary format using **np.save()** and **np.load()**. Binary files are more space-efficient and faster to read and write compared to text files.

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

np.save('array.npy', arr)
loaded_arr = np.load('array.npy')

print(loaded_arr)
```

OUTPUT:

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

6.4 Compressed File Input and Output

NumPy allows you to save and load compressed arrays using **np.savez()** and **np.load()** with the **.npz** file extension. This is useful for saving memory and reducing storage space.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

np.savez('arrays.npz', arr1=arr1, arr2=arr2)
loaded_data = np.load('arrays.npz')

print(loaded_data['arr1'])       # Output: [1 2 3]
print(loaded_data['arr2'])       # Output: [4 5 6]
```

@RAMCHANDRAPADWAL

CHAPTER N.7

Advanced NumPy Features

A Step-by-Step Guide

7.1 Broadcasting in Depth

Broadcasting is a powerful feature of NumPy that allows arrays of different shapes to be combined in arithmetic operations. NumPy automatically applies broadcasting rules to make the arrays compatible for the operation.

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([10, 20, 30])

result = arr1 + arr2
print(result)
```

OUTPUT:

```
[[11 22 33]
 [14 25 36]]
```

7.2 Array Manipulation Tricks

NumPy provides various tricks for manipulating arrays efficiently, such as swapping axes, repeating elements, and stacking arrays.

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

transposed = arr.T
repeated = np.repeat(arr, 2, axis=0)
stacked = np.stack((arr, arr), axis=0)

print(transposed)
print(repeated)
print(stacked)
```

7.3 Fancy Indexing and Boolean Masking

NumPy supports advanced indexing techniques, such as fancy indexing and boolean masking, to select specific elements or subsets of arrays based on conditions.

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

indices = np.array([0, 2, 4])
print(arr[indices])                  # Output: [1 3 5]

mask = arr > 2
print(arr[mask])                     # Output: [3 4 5]
```

7.4 Structured Arrays

NumPy allows you to create structured arrays where each element can have multiple fields with different data types. This is useful when dealing with structured data or tabular data.

```
dt = np.dtype([('name', np.str_, 16), ('age', np.int32), ('salary', np.float64)])
arr = np.array([('John', 25, 5000.0), ('Alice', 30, 6000.0)], dtype=dt)

print(arr['name'])                    # Output: ['John' 'Alice']
print(arr['age'])                     # Output: [25 30]
print(arr['salary'])                  # Output: [5000. 6000.]
```

7.5 Universal Functions

NumPy provides universal functions (ufuncs) that operate element-wise on arrays, such as add(), subtract(), multiply(), and divide(). These functions are optimized for performance.

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

print(np.add(arr, 2))              # Output: [3 4 5]
print(np.multiply(arr, 3))         # Output: [3 6 9]
print(np.sqrt(arr))                # Output: [1.      1.41421356 1.73205081]
```

CHAPTER N.8

Performance Tips and Best Practices

A Step-by-Step Guide

8.1 Vectorization

NumPy's strength lies in vectorized operations, where functions are applied to entire arrays instead of looping over individual elements. This results in faster execution and cleaner code.

```
# Non-vectorized approach
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = np.zeros(3)

for i in range(3):
    result[i] = arr1[i] + arr2[i]

print(result)                       # Output: [5. 7. 9.]

# Vectorized approach
result = arr1 + arr2
print(result)                       # Output: [5 7 9]
```

8.2 Memory Efficiency

NumPy arrays are memory-efficient compared to Python lists, especially for large datasets. However, be cautious of creating unnecessary copies of arrays, as it can consume additional memory.

```
arr1 = np.array([1, 2, 3])
arr2 = arr1

# Creating a view (no additional memory)
view = arr1.view()

# Creating a copy (additional memory)
copy = arr1.copy()
```

8.3 Use NumPy Functions Instead of Loops

NumPy provides a wide range of functions for common operations, such as sum, mean, min, max, and more. Using these functions instead of explicit loops can significantly improve performance.

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

# Non-vectorized approach
total = 0
for num in arr:
    total += num

print(total)                    # Output: 15

# Vectorized approach
total = np.sum(arr)
print(total)                    # Output: 15
```

Conclusion

NumPy is a powerful library for data science, providing efficient and flexible data structures and functions for numerical computing. In this practical guide, we covered the basics of NumPy, including array creation, indexing and slicing, array operations, array manipulation, mathematical functions, input and output, advanced features, performance tips, and best practices. With this knowledge, you can leverage NumPy to perform various data science tasks efficiently and effectively. Happy coding!