NumPy Continued and Pandas Introduction

Universal Array Functions, Axis Logic, and Series

Contents

1 Introduction

This document covers the continuation of NumPy, focusing on Universal Array Functions (ufuncs) and axis logic, followed by an introduction to Pandas Series. These concepts are fundamental for data manipulation and analysis in Python.

2 Universal Array Functions (ufuncs)

Universal functions in NumPy are functions that operate element-wise on arrays, supporting array broadcasting and type casting. They provide a fast and efficient way to perform mathematical operations on arrays.

2.1 Setup

```
import numpy as np

# Creating sample arrays for demonstration
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
c = np.array([0.1, 0.5, 0.9, 1.5])
```

Listing 1: Importing NumPy and Creating Sample Arrays

2.2 Arithmetic Functions

NumPy provides various arithmetic functions that work element-wise on arrays:

```
# Addition
np.add(a, b)  # Output: array([11, 22, 33, 44])

# Subtraction
np.subtract(a, b)  # Output: array([-9, -18, -27, -36])

# Multiplication
np.multiply(a, b)  # Output: array([10, 40, 90, 160])

# Division
np.divide(b, a)  # Output: array([10., 10., 10., 10.])

# Modulus
np.mod(b, a)  # Output: array([0, 0, 0, 0])

# Floor division
np.floor_divide(b, a)  # Output: array([10, 10, 10, 10])

# Power
np.power(a, 2)  # Output: array([1, 4, 9, 16])
```

Listing 2: Basic Arithmetic Operations

2.2.1 LCM Example

The Least Common Multiple (LCM) function can be used in different ways:

```
# LCM of two numbers
np.lcm(12, 20)  # Output: 60

```

```
# LCM of an array using reduce
arr = [4, 6, 8]
np.lcm.reduce([4, 6, 8])  # Output: 24
```

Listing 3: LCM Calculations

Note: The `reduce` method applies the LCM operation cumulatively to the elements of an array, reducing the entire array to a single value.

2.3 Comparison Functions

Comparison functions return boolean arrays indicating element-wise comparisons:

```
# Greater than
np.greater(b, a)  # Output: array([True, True, True, True])

# Less than
np.less(b, a)  # Output: array([False, False, False, False])

# Equal
np.equal(b, a)  # Output: array([False, False, False, False])

# Not equal
np.not_equal(b, a)  # Output: array([True, True, True, True])
```

Listing 4: Comparison Operations

2.4 Trigonometric Functions

NumPy provides a comprehensive set of trigonometric functions:

```
# Sine function (input in radians)
np.sin(c)  # Output: array([0.09983342, 0.47942554, 0.78332691, 0.99749499])

# Convert radians to degrees
np.degrees(1)  # Output: 57.29577951308232

# Convert degrees to radians
np.deg2rad([0, 90, 180])  # Output: array([0., 1.57079633, 3.14159265])
```

Listing 5: Trigonometric Operations

Important: Trigonometric functions in NumPy expect angles in radians, not degrees.

2.5 Exponential and Logarithmic Functions

```
# Exponential (e^x)
np.exp(a)  # Output: array([2.71828183, 7.3890561, 20.08553692, 54.59815003])

# Natural logarithm
np.log(c)  # Output: array([-2.30258509, -0.69314718, -0.10536052,
    0.40546511])

# Base-10 logarithm
np.log10(c)  # Output: array([-1., -0.30103, -0.04575749, 0.17609126])
```

Listing 6: Exponential and Logarithmic Operations

2.6 Rounding Functions

Rounding functions are essential for data preprocessing:

```
d = np.array([1.234, 2.34567, 3.4566787, 6.78899])

# Round to 2 decimal places
np.round(d, 2)  # Output: array([1.23, 2.35, 3.46, 6.79])

# Floor (round down)
np.floor(d)  # Output: array([1., 2., 3., 6.])

# Ceiling (round up)
np.ceil(d)  # Output: array([2., 3., 4., 7.])

# Truncation (remove decimal)
np.trunc(d)  # Output: array([1., 2., 3., 6.])
```

Listing 7: Rounding Operations

2.7 Statistical Functions

NumPy provides various statistical functions that can be applied to arrays:

```
# Sum of all elements
np.sum(a)  # Output: 10

# Product of all elements
np.prod(a)  # Output: 24

# Maximum value
np.max(a)  # Output: 4

# Minimum value
np.min(a)  # Output: 1

# Mean
np.mean(a)  # Output: 2.5

# Standard deviation
np.std(a)  # Output: 1.118033988749895

# Square root
np.sqrt(a)  # Output: array([1., 1.41421356, 1.73205081, 2.])

# Square
np.square(a)  # Output: array([1, 4, 9, 16])
```

Listing 8: Statistical Operations

3 Axis Logic

Understanding axis logic is crucial when working with multi-dimensional arrays. In NumPy:

- **Axis 0**: Operates along rows (vertical direction)
- **Axis 1**: Operates along columns (horizontal direction)

3.1 One-Dimensional Arrays

For 1D arrays, statistical operations work on all elements:

```
arr = np.arange(0, 10)  # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr.sum()    # 45
arr.mean()   # 4.5
arr.max()    # 9
arr.min()    # 0
arr.var()    # 8.25
arr.std()    # 2.8722813232690143
```

Listing 9: Statistical Operations on 1D Arrays

3.2 Two-Dimensional Arrays and Axis Operations

For 2D arrays, operations can be performed along specific axes:

```
arr_2d = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# Sum along axis 0 (row-wise, vertical)
arr_2d.sum(axis=0)  # array([15, 18, 21, 24])

# Sum along axis 1 (column-wise, horizontal)
arr_2d.sum(axis=1)  # array([10, 26, 42])

# Variance along axis 0
arr_2d.var(axis=0)  # array([10.66666667, 10.66666667, 10.66666667,
    10.66666667])

# Variance of all elements (no axis specified)
arr_2d.var()  # 11.916666666666666

# Variance along axis 1
arr_2d.var(axis=1)  # array([1.25, 1.25, 1.25])
```

Listing 10: Axis Operations on 2D Arrays

Key Points:

- When `axis=0`, operations are performed down the columns (across rows)
- When `axis=1`, operations are performed across the columns (along rows)
- When no axis is specified, the operation is performed on all elements

4 Introduction to Pandas

Pandas is a library for data analysis built off of NumPy. It is the backbone of data manipulation in Python and provides powerful data structures for working with structured data.

4.1 What is Pandas?

- Open-source library for Python built directly on top of NumPy
- Extremely computationally efficient
- Provides DataFrame object for tabular data

- Can handle much larger datasets than Excel (millions of rows)
- Excellent documentation available at `https://pandas.pydata.org/docs/`

4.2 Key Features of Pandas

- Tools for reading and writing data between many formats (CSV, Excel, SQL, HTML).
- Intelligently grab data based on indexing, logic, subsetting, and more.
- Handle missing data efficiently.
- Adjust and restructure data easily.

4.3 Main Topics in Pandas

The key topics we'll cover include:

- Series and DataFrames
- Conditional Filtering and Useful Methods
- Missing Data
- Group By Operations
- Combining DataFrames
- Text Methods and Time Methods
- Inputs and Outputs

4.4 Pandas Series

4.4.1 Definition

A Series is a data structure in Pandas that holds an array of information along with a named index. The named index differentiates this from a simple NumPy array.

Formal Definition: A Series is a one-dimensional ndarray with axis labels.

4.4.2 Understanding Series vs NumPy Arrays

NumPy Array with Numeric Index:

Index	Data
0	1776
1	1867
2	1821

Pandas Series with Labeled Index:

Labeled Index	Data
USA	1776
CANADA	1867
MEXICO	1821

Important Note: The data is still numerically organized internally. The Series supplements the auto-generated numeric index with a labeled index:

Numeric Index	Labeled Index	Data
0	USA	1776
1	CANADA	1867
2	MEXICO	1821

This means you can access data using either the numeric index or the labeled index.

4.4.3 Creating Series

```python
import pandas as pd
import numpy as np

# Creating from lists
labels = ['India', 'USA', 'Mexico']
my_data = [1947, 1776, 1821]

# Create Series with custom index
pd.Series(data=my_data, index=labels)
# Output:
# India      1947
# USA        1776
# Mexico     1821
# dtype: int64
```

Listing 11: Creating Pandas Series

4.4.4 Creating Series from NumPy Arrays

```python
# Create random data
names = ['Akash', 'Bob', 'Chetan', 'David']
random_data = np.random.randint(0, 100, 4)

# Create Series
pd.Series(data=random_data, index=names)
```

Listing 12: Series from NumPy Arrays

4.4.5 Creating Series from Dictionary

```python
# Dictionary automatically becomes index
ages_dict = {'Sam': 5, 'Frank': 10, 'Spike': 30}
pd.Series(ages_dict)
# Output:
# Sam        5
# Frank     10
# Spike     30
# dtype: int64
```

Listing 13: Series from Dictionary

4.5 Series Operations and Attributes

4.5.1 Creating Sales Data Example

```python
# Quarterly sales data for a company
Q1 = {'Japan': 800, 'China': 4500, 'India': 1200, 'USA': 2560}
Q2 = {'Brazil': 1100, 'China': 500, 'India': 2100, 'Mexico': 2600}

# Convert to Series
sales_Q1 = pd.Series(Q1)
sales_Q2 = pd.Series(Q2)
```

Listing 14: Sales Data Series Example

4.5.2 Accessing Series Elements

```python
# Access by label (named index)
sales_Q1['Japan']  # Returns: 800

# Access by position (deprecated, use iloc instead)
sales_Q1.iloc[0]  # Returns: 800

# Get all keys (index)
sales_Q1.keys()  # Returns: Index(['Japan', 'China', 'India', 'USA'])
```

Listing 15: Accessing Series Values

4.5.3 Series Attributes

```python
# Get index
sales_Q1.index

# Get values as NumPy array
sales_Q1.values

# Get data type
sales_Q1.dtype  # Returns: dtype('int64')

# Get shape
sales_Q1.shape  # Returns: (4,)
```

Listing 16: Series Attributes

4.6 Series Operations Between Series

When performing operations between Series with different indices, pandas aligns by index labels:

```python
# Direct addition (results in NaN for mismatched indices)
sales_Q1 + sales_Q2
# Result: NaN values where indices don't match

# Addition with fill_value for missing indices
sales_Q1.add(sales_Q2, fill_value=0)
# Missing values are treated as 0

# Broadcasting operations
sales_Q1 * 2  # Multiply all values by 2
sales_Q1 / 100  # Divide all values by 100
```

Listing 17: Series Operations with Mismatched Indices

5 DataFrames

5.1 What is a DataFrame?

A DataFrame is a table of columns and rows in pandas that we can easily restructure and filter.

Formal Definition: A DataFrame is a group of Pandas Series objects that share the same index.

5.2 From Series to DataFrame

Consider multiple Series with the same index:

Year Series

Index	Year
USA	1776
CANADA	1867
MEXICO	1821

Population Series

Index	Pop
USA	328
CANADA	38
MEXICO	126

GDP Series

Index	GDP
USA	20.5
CANADA	1.7
MEXICO	1.22

These Series with a shared index combine to form a DataFrame:

DataFrame

Index	Year	Pop	GDP
USA	1776	328	20.5
CANADA	1867	38	1.7
MEXICO	1821	126	1.22

5.3 Key Points about DataFrames

- A DataFrame is the main Pandas object we work with
- It consists of multiple Series sharing the same index
- Selecting one column returns a Series
- Selecting two or more columns returns a DataFrame
- There is no such thing as a DataFrame with one column - that's a Series
- DataFrames are extremely useful for tabular data manipulation

6 Key Takeaways

6.1 NumPy Universal Functions

- Ufuncs operate element-wise on arrays
- Support broadcasting and type casting
- Include arithmetic, comparison, trigonometric, and statistical functions
- More efficient than Python loops for array operations

6.2 Axis Logic

- Axis 0: Operations along rows (vertical direction)
- Axis 1: Operations along columns (horizontal direction)
- Critical for understanding multi-dimensional array operations
- When no axis is specified, operations apply to all elements

6.3 Pandas Series

- One-dimensional labeled array built on NumPy
- Supports both positional and labeled indexing
- Can be created from lists, NumPy arrays, or dictionaries
- Foundation for understanding DataFrames
- Operations align by index labels automatically

7 Next Steps

The next session will cover:

- DataFrames: Two-dimensional labeled data structures
- How Series combine to form DataFrames
- Advanced indexing with iloc and loc
- Data manipulation and preprocessing techniques