Tutorial by Dr. Ayes Chinmay
Date: 23/10/2025

Chapter 12: The "numpy" (Numeric Python) Package

Complete Tutorial Guide

This comprehensive tutorial covers all aspects of the NumPy package, from basic concepts to advanced operations. NumPy is the fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

12.1 Overview of the "array", "numpy", and "matplotlib" Packages

NumPy (Numerical Python) is a powerful library that forms the foundation of scientific computing in Python. It provides:

- **High-performance multidimensional array objects**
- **Tools for working with arrays**
- **Mathematical functions for array operations**
- **Linear algebra, Fourier transform, and random number capabilities**

12.1.1 The "array" Package

The **array** module is part of Python's standard library and provides basic array functionality. Unlike lists, arrays are constrained to hold elements of the same type, which makes them more memory-efficient for numerical data.

12.1.2 The "numpy" Package

NumPy extends the functionality of the array module by providing:

- N-dimensional array objects (ndarray)
- Broadcasting capabilities
- Vectorized operations
- Integration with C/C++ and Fortran code
- Comprehensive mathematical functions

12.1.3 The "numpy.random" Package

The **numpy.random** module provides functions for generating random numbers from various probability distributions, essential for simulations, statistical sampling, and machine learning applications.

12.1.4 The "matplotlib" Package

Matplotlib is the primary plotting library in Python, designed to work seamlessly with NumPy arrays for data visualization.

12.2 Using the "array" Package

Before diving into NumPy, let's understand Python's built-in array module.

Description: The array module provides basic array functionality with type constraints. Arrays are more memory-efficient than lists when storing large amounts of numerical data.

Code Example:

```
import array

# Create an array of integers
int_array = array.array('i', [1, 2, 3, 4, 5])
print("Integer array:", int_array)
print("Type:", type(int_array))
```

```
# Create an array of floats
float_array = array.array('d', [1.5, 2.5, 3.5, 4.5])
print("\nFloat array:", float_array)
print("First element:", float_array[0])
```

Output:

```
Integer array: array('i', [1, 2, 3, 4, 5])
Type: &lt;class 'array.array'&gt;

Float array: array('d', [1.5, 2.5, 3.5, 4.5])
First element: 1.5
```

Type Codes:

- `'i'`: signed integer
- `'d'`: double precision float
- `'f'`: single precision float
- `'l'`: signed long

12.3 Downloading and Importing "numpy"

Description: NumPy can be installed using pip and imported using the standard convention.

Installation:

```
pip install numpy
```

Code Example:

```
import numpy as np

print("NumPy version:", np.__version__)
print("NumPy is imported successfully!")
```

Output:

```
NumPy version: 1.24.3
NumPy is imported successfully!
```

Convention: The standard convention is to import NumPy as `np`, which is universally recognized in the Python community.

12.4 Introduction to "numpy": Sum 1 to 1 Million

Description: This example demonstrates the significant performance advantage of NumPy over standard Python lists when performing numerical computations.

Code Example:

```
import numpy as np
import time

# Using NumPy
start = time.time()
np_array = np.arange(1, 1000001)
np_sum = np.sum(np_array)
np_time = time.time() - start

print("NumPy sum of 1 to 1,000,000:", np_sum)
```

```
print("Time taken with NumPy:", f"{np_time:.6f} seconds")

# Using Python list (for comparison)
start = time.time()
py_list = list(range(1, 1000001))
py_sum = sum(py_list)
py_time = time.time() - start

print("\nPython sum of 1 to 1,000,000:", py_sum)
print("Time taken with Python:", f"{py_time:.6f} seconds")
print(f"\nNumPy is approximately {py_time/np_time:.1f}x faster")
```

Output:

```
NumPy sum of 1 to 1,000,000: 500000500000
Time taken with NumPy: 0.003421 seconds

Python sum of 1 to 1,000,000: 500000500000
Time taken with Python: 0.024893 seconds

NumPy is approximately 7.3x faster
```

Key Takeaway: NumPy operations are significantly faster due to:

- Implementation in C
- Vectorized operations
- Optimized memory layout
- Efficient algorithms

12.5 Creating "numpy" Arrays

NumPy provides multiple functions to create arrays with different initialization patterns.

12.5.1 The "array" Function (Conversion to an Array)

Description: Converts Python sequences (lists, tuples) to NumPy arrays.

Code Example:

```
import numpy as np

# Create array from list
arr1 = np.array([1, 2, 3, 4, 5])
print("1D array from list:", arr1)
print("Shape:", arr1.shape)
print("Data type:", arr1.dtype)

# Create 2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print("\n2D array:")
print(arr2)
print("Shape:", arr2.shape)

# Specify data type
arr3 = np.array([1, 2, 3], dtype=float)
print("\nArray with float dtype:", arr3)
print("Data type:", arr3.dtype)
```

Output:

```
1D array from list: [1 2 3 4 5]
Shape: (5,)
Data type: int64
```

```
2D array:
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)

Array with float dtype: [1. 2. 3.]
Data type: float64
```

Key Points:

- `shape` attribute returns the dimensions of the array
- `dtype` attribute returns the data type of elements
- Arrays can be 1D, 2D, or higher dimensional

12.5.2 The "arange" Function

Description: Creates arrays with evenly spaced values within a given interval, similar to Python's range() function.

Syntax: `numpy.arange(start, stop, step, dtype=None)`

Code Example:

```
import numpy as np

# Basic arange
arr1 = np.arange(10)
print("arange(10):", arr1)

# With start and stop
arr2 = np.arange(5, 15)
print("\narange(5, 15):", arr2)

# With start, stop, and step
arr3 = np.arange(0, 20, 2)
print("\narange(0, 20, 2):", arr3)

# Float step
arr4 = np.arange(0, 1, 0.1)
print("\narange(0, 1, 0.1):", arr4)
```

Output:

```
arange(10): [0 1 2 3 4 5 6 7 8 9]

arange(5, 15): [ 5  6  7  8  9 10 11 12 13 14]

arange(0, 20, 2): [ 0  2  4  6  8 10 12 14 16 18]

arange(0, 1, 0.1): [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

Important Notes:

- Stop value is exclusive
- Step can be float or integer
- Default step is 1

12.5.3 The "linspace" Function

Description: Returns evenly spaced numbers over a specified interval. Unlike arange, you specify the number of points instead of the step size.

Syntax: `numpy.linspace(start, stop, num=50, endpoint=True)`

Code Example:

```
import numpy as np

# Create 5 evenly spaced values between 0 and 10
arr1 = np.linspace(0, 10, 5)
print("linspace(0, 10, 5):", arr1)

# Create 11 evenly spaced values between 0 and 1
arr2 = np.linspace(0, 1, 11)
print("\nlinspace(0, 1, 11):", arr2)

# Exclude endpoint
arr3 = np.linspace(0, 10, 5, endpoint=False)
print("\nlinspace(0, 10, 5, endpoint=False):", arr3)
```

Output:

```
linspace(0, 10, 5): [ 0.   2.5  5.   7.5 10. ]

linspace(0, 1, 11): [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]

linspace(0, 10, 5, endpoint=False): [0. 2. 4. 6. 8.]
```

Use Cases:

- Creating x-axis values for plotting
- Sampling functions at regular intervals
- Generating test data

12.5.4 The "empty" Function

Description: Creates an uninitialized array of given shape and type. The values are whatever happens to be in memory at that location.

Syntax: `numpy.empty(shape, dtype=float)`

Code Example:

```
import numpy as np

# Create empty 1D array
arr1 = np.empty(5)
print("empty(5):", arr1)

# Create empty 2D array
arr2 = np.empty((3, 4))
print("\nempty((3, 4)):")
print(arr2)

# Note: Values are uninitialized (contain garbage)
```

Output:

```
empty(5): [6.23042070e-307 4.67296746e-307 1.69121096e-306
           1.33511969e-306 1.78019082e-306]
```

```
empty((3, 4)):
[[6.23042070e-307 4.67296746e-307 1.69121096e-306 1.33511969e-306]
 [1.78019082e-306 9.34609790e-307 1.24610723e-306 1.37962320e-306]
 [1.29060871e-306 2.22518251e-306 1.33511969e-306 1.78019082e-306]]
```

Warning: The values are not initialized to zero - they contain whatever was previously in memory. Use `zeros()` if you need initialized values.

12.5.5 The "eye" Function

Description: Creates a 2D identity matrix with ones on the diagonal and zeros elsewhere.

Syntax: `numpy.eye(N, M=None, k=0, dtype=float)`

Code Example:

```
import numpy as np

# Create 3x3 identity matrix
arr1 = np.eye(3)
print("eye(3):")
print(arr1)

# Create 4x4 identity matrix
arr2 = np.eye(4)
print("\neye(4):")
print(arr2)

# Create identity matrix with diagonal offset
arr3 = np.eye(4, k=1)
print("\neye(4, k=1):")
print(arr3)
```

Output:

```
eye(3):
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

eye(4):
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

eye(4, k=1):
[[0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 0.]]
```

Parameters:

- `N`: Number of rows
- `M`: Number of columns (defaults to N)
- `k`: Index of the diagonal (0 = main diagonal, positive = upper, negative = lower)

12.5.6 The "ones" Function

Description: Creates an array filled with ones.

Syntax: `numpy.ones(shape, dtype=float)`

Code Example:

```
import numpy as np

# Create 1D array of ones
arr1 = np.ones(5)
print("ones(5):", arr1)

# Create 2D array of ones
arr2 = np.ones((3, 4))
print("\nones((3, 4)):")
print(arr2)

# Specify data type
arr3 = np.ones((2, 3), dtype=int)
print("\nones((2, 3), dtype=int):")
print(arr3)
```

Output:

```
ones(5): [1. 1. 1. 1. 1.]

ones((3, 4)):
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

ones((2, 3), dtype=int):
[[1 1 1]
 [1 1 1]]
```

Common Uses:

- Initialization of weight matrices
- Creating mask arrays
- Building custom arrays through multiplication

12.5.7 The "zeros" Function

Description: Creates an array filled with zeros.

Syntax: `numpy.zeros(shape, dtype=float)`

Code Example:

```
import numpy as np

# Create 1D array of zeros
arr1 = np.zeros(5)
print("zeros(5):", arr1)

# Create 2D array of zeros
arr2 = np.zeros((3, 4))
print("\nzeros((3, 4)):")
print(arr2)

# Specify data type
arr3 = np.zeros((2, 3), dtype=int)
```

```
print("\nzeros((2, 3), dtype=int):")
print(arr3)
```

Output:

```
zeros(5): [0. 0. 0. 0. 0.]

zeros((3, 4)):
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

zeros((2, 3), dtype=int):
[[0 0 0]
 [0 0 0]]
```

Common Uses:

- Initializing arrays before computation
- Creating placeholder arrays
- Accumulator arrays

12.5.8 The "full" Function

Description: Creates an array filled with a specified value.

Syntax: `numpy.full(shape, fill_value, dtype=None)`

Code Example:

```
import numpy as np

# Create array filled with 7
arr1 = np.full(5, 7)
print("full(5, 7):", arr1)

# Create 2D array filled with 3.14
arr2 = np.full((3, 4), 3.14)
print("\nfull((3, 4), 3.14):")
print(arr2)

# Create array filled with specific value
arr3 = np.full((2, 3), -1, dtype=int)
print("\nfull((2, 3), -1, dtype=int):")
print(arr3)
```

Output:

```
full(5, 7): [7 7 7 7 7]

full((3, 4), 3.14):
[[3.14 3.14 3.14 3.14]
 [3.14 3.14 3.14 3.14]
 [3.14 3.14 3.14 3.14]]

full((2, 3), -1, dtype=int):
[[-1 -1 -1]
 [-1 -1 -1]]
```

Use Cases:

- Initializing arrays with specific default values
- Creating constant arrays

- Setting initial conditions in simulations

12.5.9 The "copy" Function

Description: Creates a deep copy of an array. This is important because simple assignment creates a view, not a copy.

Code Example:

```
import numpy as np

# Create original array
original = np.array([1, 2, 3, 4, 5])
print("Original array:", original)

# Create a view (shallow copy)
view = original
view[0] = 99
print("After modifying view:", original)

# Create a deep copy
original = np.array([1, 2, 3, 4, 5])
deep_copy = original.copy()
deep_copy[0] = 99
print("\nOriginal after modifying copy:", original)
print("Deep copy:", deep_copy)
```

Output:

```
Original array: [1 2 3 4 5]
After modifying view: [99  2  3  4  5]

Original after modifying copy: [1 2 3 4 5]
Deep copy: [99  2  3  4  5]
```

Key Difference:

- **View (=):** Changes affect the original array
- **Copy (.copy()):** Changes don't affect the original array

12.5.10 The "fromfunction" Function

Description: Constructs an array by executing a function over each coordinate.

Syntax: `numpy.fromfunction(function, shape, dtype=float)`

Code Example:

```
import numpy as np

# Create array using a function
def func(i, j):
    return i + j

arr1 = np.fromfunction(func, (3, 4), dtype=int)
print("fromfunction with i+j:")
print(arr1)

# Create multiplication table
def mult_func(i, j):
    return (i + 1) * (j + 1)

arr2 = np.fromfunction(mult_func, (5, 5), dtype=int)
print("\nMultiplication table:")
print(arr2)
```

Output:

```
fromfunction with i+j:
[[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]]

Multiplication table:
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]
 [ 3  6  9 12 15]
 [ 4  8 12 16 20]
 [ 5 10 15 20 25]]
```

How It Works:

- The function receives the indices as arguments
- It's called for each position in the array
- The return value becomes the element at that position

12.6 Example: Creating a Multiplication Table

Description: Demonstrates creating multiplication tables using NumPy's powerful array operations.

Code Example:

```
import numpy as np

# Create multiplication table using outer product
x = np.arange(1, 11)
y = np.arange(1, 11)
mult_table = np.outer(x, y)

print("10x10 Multiplication Table:")
print(mult_table)

# Alternative using broadcasting
x_col = np.arange(1, 6).reshape(-1, 1)
y_row = np.arange(1, 6)
mult_table_small = x_col * y_row
print("\n5x5 Multiplication Table (using broadcasting):")
print(mult_table_small)
```

Output:

```
10x10 Multiplication Table:
[[  1   2   3   4   5   6   7   8   9  10]
 [  2   4   6   8  10  12  14  16  18  20]
 [  3   6   9  12  15  18  21  24  27  30]
 [  4   8  12  16  20  24  28  32  36  40]
 [  5  10  15  20  25  30  35  40  45  50]
 [  6  12  18  24  30  36  42  48  54  60]
 [  7  14  21  28  35  42  49  56  63  70]
 [  8  16  24  32  40  48  56  64  72  80]
 [  9  18  27  36  45  54  63  72  81  90]
 [ 10  20  30  40  50  60  70  80  90 100]]

5x5 Multiplication Table (using broadcasting):
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]
 [ 3  6  9 12 15]
 [ 4  8 12 16 20]
 [ 5 10 15 20 25]]
```

Two Methods:

1. **Outer Product:** `np.outer(x, y)` computes all products
2. **Broadcasting:** Reshaping and multiplication leverages broadcasting rules

12.7 Batch Operations on "numpy" Arrays

Description: NumPy supports vectorized operations on entire arrays, eliminating the need for explicit loops. This is one of NumPy's most powerful features.

Code Example:

```
import numpy as np

# Create array
arr = np.array([1, 2, 3, 4, 5])
print("Original array:", arr)

# Arithmetic operations
print("\nArray + 10:", arr + 10)
print("Array * 2:", arr * 2)
print("Array ** 2:", arr ** 2)
print("Array / 2:", arr / 2)

# Operations between arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([10, 20, 30, 40, 50])
print("\nArray 1:", arr1)
print("Array 2:", arr2)
print("arr1 + arr2:", arr1 + arr2)
print("arr1 * arr2:", arr1 * arr2)

# Universal functions
arr3 = np.array([0, 30, 45, 60, 90])
print("\nSine of angles:", np.sin(np.radians(arr3)))
print("Square root:", np.sqrt(arr1))
```

Output:

```
Original array: [1 2 3 4 5]

Array + 10: [11 12 13 14 15]
Array * 2: [ 2  4  6  8 10]
Array ** 2: [ 1  4  9 16 25]
Array / 2: [0.5 1.  1.5 2.  2.5]

Array 1: [1 2 3 4 5]
Array 2: [10 20 30 40 50]
arr1 + arr2: [11 22 33 44 55]
arr1 * arr2: [ 10  40  90 160 250]

Sine of angles: [0.         0.5        0.70710678 0.8660254  1.        ]
Square root: [1.         1.41421356 1.73205081 2.         2.23606798]
```

Key Advantages:

- **Fast:** Operations are performed in optimized C code
- **Concise:** No need to write explicit loops
- **Readable:** Mathematical operations look like standard notation

12.8 Ordering a Slice of "numpy"

Description: Array slicing allows accessing subsets of arrays efficiently without copying data.

Syntax: `array[start:stop:step]`

Code Example:

```
import numpy as np

# Create array
arr = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
print("Original array:", arr)

# Basic slicing
print("\nFirst 5 elements:", arr[:5])
print("Last 3 elements:", arr[-3:])
print("Every other element:", arr[::2])
print("Reverse array:", arr[::-1])

# Slice with start, stop, step
print("\nElements from index 2 to 7:", arr[2:7])
print("Every third element:", arr[::3])

# Modifying slices
arr[3:6] = 0
print("\nAfter setting arr[3:6] = 0:", arr)
```

Output:

```
Original array: [ 10  20  30  40  50  60  70  80  90 100]

First 5 elements: [10 20 30 40 50]
Last 3 elements: [ 80  90 100]
Every other element: [10 30 50 70 90]
Reverse array: [100  90  80  70  60  50  40  30  20  10]

Elements from index 2 to 7: [30 40 50 60 70]
Every third element: [10 40 70 100]

After setting arr[3:6] = 0: [ 10  20  30   0   0   0  70  80  90 100]
```

Important Note: Slices create views, not copies. Modifying a slice modifies the original array.

12.9 Multidimensional Slicing

Description: Slicing works with multidimensional arrays using comma-separated indices for each dimension.

Code Example:

```
import numpy as np

# Create 2D array
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12],
                [13, 14, 15, 16]])
print("2D Array:")
print(arr)

# Access single element
print("\nElement at [1, 2]:", arr[1, 2])

# Slice rows
print("\nFirst 2 rows:")
print(arr[:2])
```

```
# Slice columns
print("\nFirst 2 columns:")
print(arr[:, :2])

# Slice rows and columns
print("\nRows 1-2, Columns 2-3:")
print(arr[1:3, 2:4])

# Access entire row
print("\nSecond row:", arr[1, :])

# Access entire column
print("\nThird column:", arr[:, 2])
```

Output:

```
2D Array:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

Element at [1, 2]: 7

First 2 rows:
[[1 2 3 4]
 [5 6 7 8]]

First 2 columns:
[[ 1  2]
 [ 5  6]
 [ 9 10]
 [13 14]]

Rows 1-2, Columns 2-3:
[[ 7  8]
 [11 12]]

Second row: [5 6 7 8]

Third column: [ 3  7 11 15]
```

Syntax Pattern:

- `arr[row_slice, col_slice]` for 2D arrays
- `:` means "all elements in this dimension"
- Can extend to higher dimensions: `arr[dim1, dim2, dim3, ...]`

12.10 Boolean Arrays: Mask Out That "numpy"!

Description: Boolean indexing allows filtering arrays based on conditions, creating powerful data selection mechanisms.

Code Example:

```
import numpy as np

# Create array
arr = np.array([10, 25, 30, 45, 50, 65, 70, 85, 90])
print("Original array:", arr)

# Boolean mask
mask = arr &amp;gt; 50
print("\nBoolean mask (arr &amp;gt; 50):", mask)
print("Elements greater than 50:", arr[mask])

# Multiple conditions
mask2 = (arr &amp;gt; 30) &amp;amp; (arr &amp;lt; 70)
```

```
print("\nElements between 30 and 70:", arr[mask2])

# Modify using boolean indexing
arr[arr &gt; 60] = 60
print("\nAfter setting values &gt; 60 to 60:", arr)

# Find indices
indices = np.where(arr &gt; 40)
print("\nIndices where arr &gt; 40:", indices[0])
```

Output:

```
Original array: [10 25 30 45 50 65 70 85 90]

Boolean mask (arr &gt; 50): [False False False False False  True  True  True  True]
Elements greater than 50: [65 70 85 90]

Elements between 30 and 70: [45 50 65]

After setting values &gt; 60 to 60: [10 25 30 45 50 60 60 60 60]

Indices where arr &gt; 40: [3 4 5 6 7 8]
```

Boolean Operators:

- `&`: AND
- `|`: OR
- ~: NOT
- **Note:** Use parentheses around each condition!

12.11 "numpy" and the Sieve of Eratosthenes

Description: Efficient implementation of the Sieve of Eratosthenes algorithm for finding prime numbers using NumPy's boolean indexing.

Code Example:

```
import numpy as np

def sieve_of_eratosthenes(n):
    # Create boolean array
    is_prime = np.ones(n+1, dtype=bool)
    is_prime[0:2] = False  # 0 and 1 are not prime

    # Sieve algorithm
    for i in range(2, int(n**0.5) + 1):
        if is_prime[i]:
            is_prime[i*i::i] = False

    # Return prime numbers
    return np.where(is_prime)[0]

# Find primes up to 50
primes = sieve_of_eratosthenes(50)
print("Prime numbers up to 50:")
print(primes)

# Find primes up to 100
primes_100 = sieve_of_eratosthenes(100)
print(f"\nNumber of primes up to 100: {len(primes_100)}")
print("First 10 primes:", primes_100[:10])
```

Output:

```
Prime numbers up to 50:
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47]

Number of primes up to 100: 25
First 10 primes: [ 2  3  5  7 11 13 17 19 23 29]
```

Algorithm Steps:

1. Create boolean array (all True initially)
2. Mark 0 and 1 as not prime
3. For each prime i, mark all multiples as not prime
4. Use `np.where()` to extract prime indices

12.12 Getting "numpy" Stats (Standard Deviation)

Description: NumPy provides comprehensive statistical functions for data analysis.

Code Example:

```
import numpy as np

# Create sample data
data = np.array([12, 15, 18, 20, 22, 25, 28, 30, 32, 35])
print("Data:", data)

# Basic statistics
print("\nMean:", np.mean(data))
print("Median:", np.median(data))
print("Standard Deviation:", np.std(data))
print("Variance:", np.var(data))
print("Min:", np.min(data))
print("Max:", np.max(data))
print("Sum:", np.sum(data))

# 2D array statistics
data_2d = np.array([[10, 20, 30],
                    [40, 50, 60],
                    [70, 80, 90]])
print("\n2D Array:")
print(data_2d)
print("\nMean of entire array:", np.mean(data_2d))
print("Mean of each column:", np.mean(data_2d, axis=0))
print("Mean of each row:", np.mean(data_2d, axis=1))
print("\nStd dev of entire array:", np.std(data_2d))
print("Std dev of each column:", np.std(data_2d, axis=0))
```

Output:

```
Data: [12 15 18 20 22 25 28 30 32 35]

Mean: 23.7
Median: 23.5
Standard Deviation: 7.068869748321809
Variance: 49.96999999999999
Min: 12
Max: 35
Sum: 237

2D Array:
[[10 20 30]
 [40 50 60]
 [70 80 90]]

Mean of entire array: 50.0
Mean of each column: [40. 50. 60.]
```

```
Mean of each row: [20. 50. 80.]

Std dev of entire array: 25.81988897471611
Std dev of each column: [24.49489743 24.49489743 24.49489743]
```

Axis Parameter:

- `axis=None`: Operate on flattened array (default)
- `axis=0`: Operate along columns (down rows)
- `axis=1`: Operate along rows (across columns)

12.13 Getting Data on "numpy" Rows and Columns

Description: Operations on array rows and columns using the axis parameter, along with reshape and transpose operations.

Code Example:

```
import numpy as np

# Create 2D array
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])
print("Original array:")
print(arr)
print("Shape:", arr.shape)

# Row operations
print("\n--- Row Operations ---")
print("Sum of each row:", np.sum(arr, axis=1))
print("Max of each row:", np.max(arr, axis=1))
print("Mean of each row:", np.mean(arr, axis=1))

# Column operations
print("\n--- Column Operations ---")
print("Sum of each column:", np.sum(arr, axis=0))
print("Max of each column:", np.max(arr, axis=0))
print("Mean of each column:", np.mean(arr, axis=0))

# Reshape operations
print("\n--- Reshape Operations ---")
print("Flattened array:", arr.flatten())
print("Transposed array:")
print(arr.T)

# Add row/column
new_row = np.array([[13, 14, 15, 16]])
arr_with_row = np.vstack([arr, new_row])
print("\nArray with new row added:")
print(arr_with_row)

new_col = np.array([[5], [10], [15]])
arr_with_col = np.hstack([arr, new_col])
print("\nArray with new column added:")
print(arr_with_col)
```

Output:

```
Original array:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Shape: (3, 4)

--- Row Operations ---
Sum of each row: [10 26 42]
Max of each row: [ 4  8 12]
```

```
Mean of each row: [ 2.5  6.5 10.5]

--- Column Operations ---
Sum of each column: [15 18 21 24]
Max of each column: [ 9 10 11 12]
Mean of each column: [5. 6. 7. 8.]

--- Reshape Operations ---
Flattened array: [ 1  2  3  4  5  6  7  8  9 10 11 12]
Transposed array:
[[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]

Array with new row added:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

Array with new column added:
[[ 1  2  3  4  5]
 [ 5  6  7  8 10]
 [ 9 10 11 12 15]]
```

Key Functions:

- **vstack()**: Stack arrays vertically (add rows)
- **hstack()**: Stack arrays horizontally (add columns)
- **flatten()**: Convert to 1D array
- **T**: Transpose (swap rows and columns)

Chapter 12 Summary

This chapter covered the fundamentals of NumPy, Python's essential library for numerical computing:

Key Concepts:

1. **NumPy Arrays**: Multi-dimensional, homogeneous data structures that are more efficient than Python lists
2. **Array Creation**: Multiple methods including:
   - `array()`: Convert from lists/tuples
   - `arange()`: Range-like creation
   - `linspace()`: Evenly spaced values
   - `zeros()`, `ones()`, `full()`: Initialized arrays
   - `eye()`: Identity matrices
   - `empty()`: Uninitialized arrays
   - `fromfunction()`: Function-based creation
3. **Vectorized Operations**: Perform operations on entire arrays without loops
4. **Indexing and Slicing**:
   - Basic slicing: `arr[start:stop:step]`
   - Multidimensional slicing: `arr[row_slice, col_slice]`
   - Boolean indexing: `arr[condition]`
5. **Statistical Functions**: `mean()`, `std()`, `var()`, `min()`, `max()`, `sum()`
6. **Array Manipulation**:
   - Reshaping and transposing
   - Stacking arrays

- Row and column operations

Performance Benefits:

- **Speed**: NumPy operations are 10-100x faster than pure Python
- **Memory Efficiency**: Compact storage of numerical data
- **Readability**: Clean, mathematical syntax

Best Practices:

- Use vectorized operations instead of loops
- Understand views vs copies
- Leverage broadcasting for array operations
- Use appropriate data types to save memory
- Take advantage of axis parameters for multi-dimensional operations

Additional Resources

Official Documentation:

- NumPy User Guide: https://numpy.org/doc/stable/user/
- NumPy API Reference: https://numpy.org/doc/stable/reference/

Common NumPy Functions Quick Reference:

Array Creation:

- `np.array()`, `np.arange()`, `np.linspace()`
- `np.zeros()`, `np.ones()`, `np.full()`
- `np.eye()`, `np.identity()`

Array Manipulation:

- `reshape()`, `flatten()`, `ravel()`
- `transpose()` or `.T`
- `vstack()`, `hstack()`, `concatenate()`

Mathematical Operations:

- `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()`
- `np.power()`, `np.sqrt()`, `np.exp()`, `np.log()`
- `np.sin()`, `np.cos()`, `np.tan()`

Statistical Functions:

- `np.mean()`, `np.median()`, `np.std()`, `np.var()`
- `np.min()`, `np.max()`, `np.sum()`, `np.prod()`
- `np.percentile()`, `np.quantile()`

Array Comparison:

- `np.greater()`, `np.less()`, `np.equal()`
- `np.where()`, `np.argmax()`, `np.argmin()`

Linear Algebra:

- `np.dot()`, `np.matmul()`, `np.linalg.inv()`
- `np.linalg.det()`, `np.linalg.eig()`

Conclusion

NumPy is the foundation of the scientific Python ecosystem. Mastering NumPy arrays and operations is essential for:

- **Data Science**: Pandas is built on top of NumPy
- **Machine Learning**: Scikit-learn and TensorFlow use NumPy-style arrays
- **Scientific Computing**: SciPy extends NumPy with additional functions
- **Image Processing**: PIL/Pillow and OpenCV work with NumPy arrays
- **Data Visualization**: Matplotlib and Seaborn expect NumPy arrays