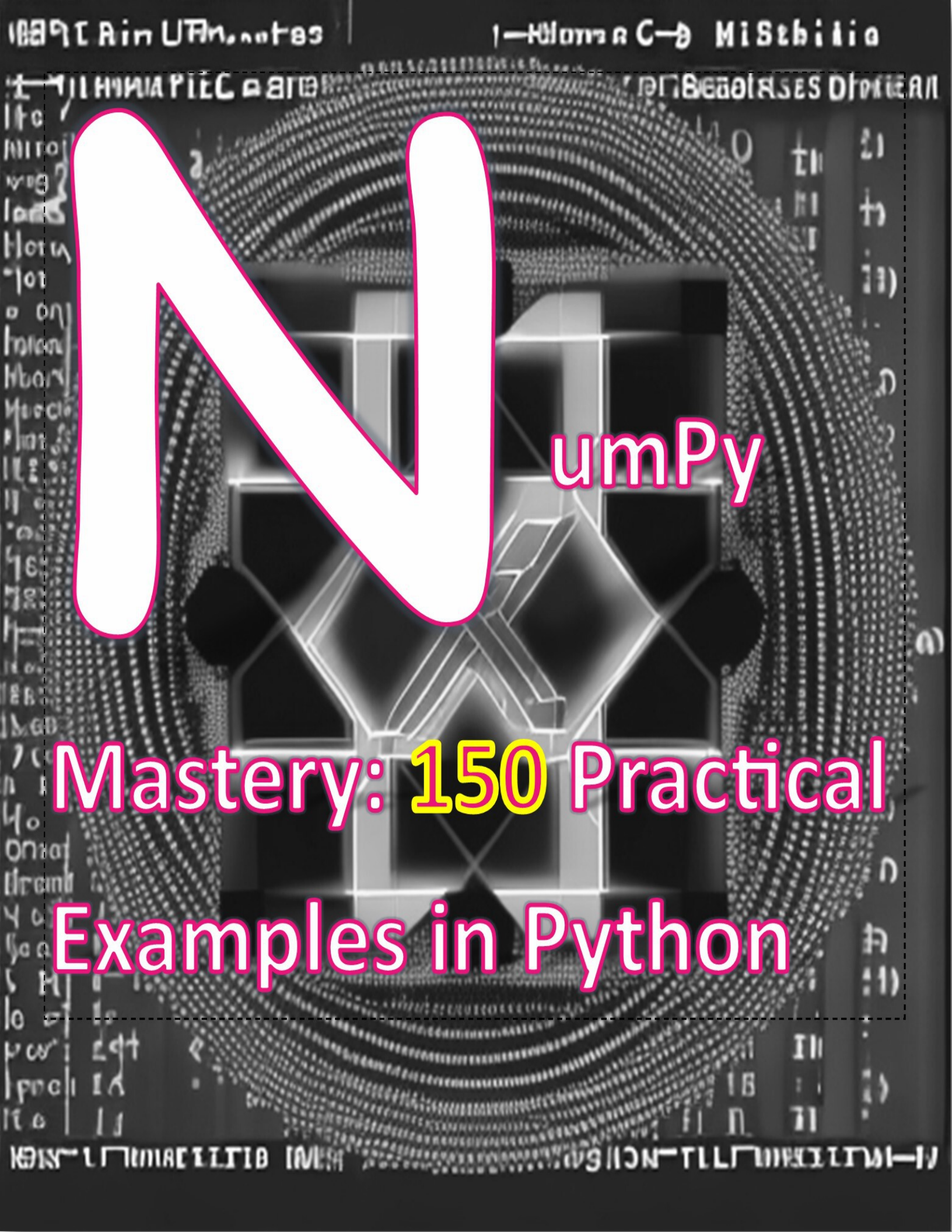
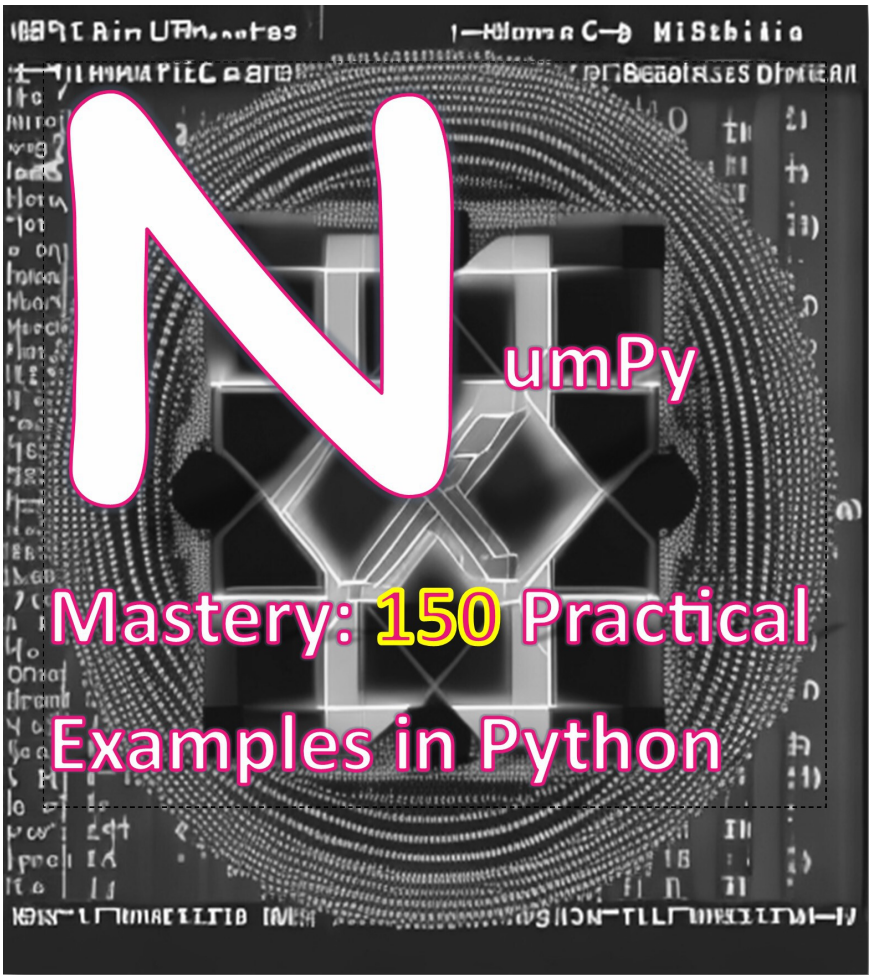

NumPy
Mastery: 150 Practical
Examples in Python


NumPy
Mastery: 150 Practical
Examples in Python

Understanding NumPy Universal Functions (ufuncs)

NumPy, short for Numerical Python, is a fundamental library in Python for numerical computations . It provides support for working with arrays and matrices of data efficiently . Universal Functions, or ufuncs, are at the core of NumPy's power, enabling element - wise operations on arrays . This deep dive explores ufuncs in greater detail, covering their creation, different categories, and specific use cases .

Introduction to ufuncs

Ufuncs are a key feature of NumPy, designed to perform element - wise operations on arrays with remarkable efficiency . At their core, ufuncs are a type of function implemented in C that can operate on arrays, broadcasting the operation if necessary . These functions are designed to work seamlessly with arrays of various sizes and dimensions, making them a versatile tool for mathematical and numerical operations .

Key Advantages of ufuncs

Ufuncs offer several notable advantages :

- **Speed** : Ufuncs are highly optimized, often faster than equivalent Python code with loops . The C implementation ensures efficient computation .
- **Convenience** : They provide a concise and convenient way to express mathematical operations on arrays, eliminating the need for explicit loops .
- **Broadcasting** : Ufuncs automatically handle broadcasting, allowing operations on arrays of different shapes, which greatly simplifies code .

Creating Custom ufuncs

While NumPy offers a wide range of built - in ufuncs for common operations, you can also create custom ufuncs . This allows you to define your own element - wise functions and apply them to arrays efficiently .

Example : Creating a Custom ufunc

pythonCopy code

```
import numpy as np

# Define a custom Python function
def custom_function( x ):
    return x ** 2  + 2  * x + 1

# Create a ufunc from the custom function
custom_ufunc = np.frompyfunc(custom_function, 1, 1)

# Apply the custom ufunc to an array
arr = np.array([1, 2, 3, 4])
result = custom_ufunc(arr)

print(result)   # Output : [ 4 9 16 25 ]
```

In this example, we define a custom function custom_function and create a ufunc custom_ufunc from it . The ufunc is then applied to an array, performing element - wise operations .

Simple Arithmetic with ufuncs

Ufuncs provide a wide array of basic arithmetic operations, including addition, subtraction, multiplication, and division, among others . These operations are essential for numerical computations and are optimized for performance .

Example : Performing Arithmetic Operations with ufuncs

pythonCopy code

```
import numpy as np

# Create two arrays
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

# Perform arithmetic operations using ufuncs
addition_result = np.add(arr1, arr2)
subtraction_result = np.subtract(arr1, arr2)
multiplication_result = np.multiply(arr1, arr2)
division_result = np.divide(arr1, arr2)
```

```
print("Addition:", addition_result)
print("Subtraction:", subtraction_result)
print("Multiplication:", multiplication_result)
print("Division:", division_result)
```

In this example, we utilize ufuncs like np.add() , np.subtract() , np.multiply() , and np.divide() to perform element - wise arithmetic operations on two arrays .

Rounding Decimals with ufuncs

Ufuncs provide functions to round and truncate decimal values within arrays . This is crucial when working with data that requires specific precision or formatting .

Example : Rounding Decimals with ufuncs

pythonCopy code

```
import numpy as np

# Create an array with decimal values
arr = np.array([1.23, 2.45, 3.67, 4.89])

# Round elements to the nearest integer
rounded_arr = np.round(arr)

# Round down to the nearest integer
floor_arr = np.floor(arr)

# Round up to the nearest integer
ceil_arr = np.ceil(arr)

print("Rounded:", rounded_arr)
print("Floor:", floor_arr)
print("Ceil:", ceil_arr)
```

In this example, we employ ufuncs like np.round() , np.floor() , and np.ceil() to round and truncate decimal values in an array .

Logarithmic Operations with ufuncs

NumPy's ufuncs include functions for both natural logarithms (base e) and common logarithms (base 10). These are vital for working with data that follows exponential patterns, such as financial data or scientific measurements .

Example : Logarithmic Operations with ufuncs

pythonCopy code

```
import numpy as np

# Create an array of positive values
arr = np.array([1, 10, 100, 1000])

# Calculate natural logarithms ( ln ) of elements
ln_arr = np.log(arr)

# Calculate base 10 logarithms of elements
log10_arr = np.log10(arr)

print("Natural Logarithm (ln):", ln_arr)
print("Base 10 Logarithm:", log10_arr)
```

In this example, we use ufuncs like np.log() and np.log10() to compute the natural logarithm and base 10 logarithm of array elements, respectively .

Summations with ufuncs

Ufuncs offer efficient methods for calculating sums of array elements, both individually and cumulatively . These operations are essential for various applications, including statistics and data analysis .

Example : Summing Array Elements with ufuncs

pythonCopy code

```
import numpy as np

# Create an array of numbers
arr = np.array([1, 2, 3, 4, 5])

# Calculate the sum of all elements
total_sum = np.sum(arr)

# Calculate the cumulative sum
cumulative_sum = np.cumsum(arr)
```

```
print("Total Sum:", total_sum)
print("Cumulative Sum:", cumulative_sum)
```

In this example, we use the np.sum() ufunc to find the total sum of elements in an array and the np.cumsum() ufunc to calculate the cumulative sum .

Products with ufuncs

Ufuncs provide efficient ways to compute the product of array elements, both individually and cumulatively . Product operations are commonly used in mathematical calculations and statistical analysis .

Example : Computing Products with ufuncs

pythonCopy code

```
import numpy as np

# Create an array of numbers
arr = np.array([1, 2, 3, 4, 5])

# Calculate the product of all elements
total_product = np.prod(arr)

# Calculate the cumulative product
cumulative_product = np.cumprod(arr)

print("Total Product:", total_product)
print("Cumulative Product:", cumulative_product)
```

In this example, we use the np.prod() ufunc to find the total product of elements in an array and the np.cumprod() ufunc to calculate the cumulative product .

Differences with ufuncs

Ufuncs provide efficient methods for computing the differences between elements in an array, both individually and cumulatively . These operations are valuable in fields such as time series analysis and signal processing .

Example : Calculating Differences with ufuncs

pythonCopy code

```
import numpy as np
```

```
# Create an array of numbers
arr = np.array([1, 2, 4, 7, 11])

# Calculate the differences between adjacent elements
differences = np.diff(arr)

print("Differences:", differences)
```

In this example, we use the np.diff() ufunc to calculate the differences between adjacent elements in the array .

Finding the Least Common Multiple (LCM) with ufuncs

Ufuncs in NumPy can be used to find the Least Common Multiple (LCM) of multiple integers . The LCM is the smallest positive integer that is divisible by each of the given integers .

Example : Finding the LCM with ufuncs

pythonCopy code

```
import numpy as np

# Create an array of integers
integers = np.array([12, 18, 24])

# Find the LCM of the integers
lcm = np.lcm.reduce(integers)

print("LCM:", lcm)
```

In this example, we use the np.lcm.reduce() ufunc to find the LCM of the integers in the array .

Finding the Greatest Common Divisor (GCD) with ufuncs

Ufuncs in NumPy can also be used to find the Greatest Common Divisor (GCD) of multiple integers . The GCD is the largest positive integer that divides each of the given integers without a remainder .

Example : Finding the GCD with ufuncs

pythonCopy code

```
import numpy as np

# Create an array of integers
integers = np.array([18, 24, 36])
```

```
# Find the GCD of the integers
gcd = np.gcd.reduce(integers)

print("GCD:", gcd)
```

In this example, we use the np.gcd.reduce() ufunc to find the GCD of the integers in the array .

Trigonometric Operations with ufuncs

NumPy's ufuncs include a wide range of trigonometric functions for working with angles and trigonometric identities . These functions are vital in various scientific and engineering applications .

Example : Trigonometric Operations with ufuncs

```
pythonCopy code
import numpy as np

# Create an array of angles in radians
angles_radians = np.array([0, np.pi/4, np.pi/2, np.pi])

# Calculate sine, cosine, and tangent of the angles
sine = np.sin(angles_radians)
cosine = np.cos(angles_radians)
tangent = np.tan(angles_radians)

print("Sine:", sine)
print("Cosine:", cosine)
print("Tangent:", tangent)
```

In this example, we use ufuncs like np.sin() , np.cos() , and np.tan() to compute the sine, cosine, and tangent of angles in radians .

Hyperbolic Operations with ufuncs

Ufuncs in NumPy also provide hyperbolic trigonometric functions for working with hyperbolic identities . These functions are essential in fields like physics, engineering, and statistics .

Example : Hyperbolic Operations with ufuncs

pythonCopy code

```
import numpy as np

# Create an array of values
values = np.array([0, 1, 2])

# Calculate hyperbolic sine and cosine
sinh_values = np.sinh(values)
cosh_values = np.cosh(values)

print("Hyperbolic Sine:", sinh_values)
print("Hyperbolic Cosine:", cosh_values)
```

In this example, we use ufuncs like np.sinh() and np.cosh() to calculate hyperbolic sine and cosine .

Set Operations with ufuncs

NumPy's ufuncs include functions for performing set operations on arrays, such as union, intersection, and set difference . These operations are crucial when dealing with data that can be categorized or grouped .

Example : Set Operations with ufuncs

pythonCopy code

```
import numpy as np

# Create two arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([3, 4, 5, 6, 7])

# Find the union of the two arrays
union = np.union1d(arr1, arr2)

# Find the intersection of the two arrays
intersection = np.intersect1d(arr1, arr2)

# Find the set difference ( elements in arr1 but not in arr2 )
difference = np.setdiff1d(arr1, arr2)

print("Union:", union)
print("Intersection:", intersection)
print("Set Difference:", difference)
```

In this example, we use ufuncs like np.union1d() , np.intersect1d() , and np.setdiff1d() to perform set operations on two arrays .

This in - depth exploration of NumPy Universal Functions (ufuncs) provides a solid foundation for leveraging their power in various scientific, engineering, and data analysis tasks . By understanding the breadth of operations and the efficiency ufuncs offer, you can optimize your Python code for complex numerical computations and data manipulation .

150 examples of how you can use NumPy in Python for various tasks :

1 . Import NumPy :

```
import numpy as np
```

2 . Create a 1D NumPy array :

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

3 . Create a 2D NumPy array :

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

4 . Get the shape of an array :

```
shape = arr.shape
```

5 . Reshape an array :

```
reshaped_arr = arr.reshape(2, 3)
```

6 . Generate an array of zeros :

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

7 . Generate an array of ones :

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

8 . Generate an identity matrix :

```
identity = np.eye(3)
```

9 . Create a NumPy array filled with a specific value :

```
filled_arr = np.full((2, 2), 7)
```

10 . Generate a range of numbers using arange:

```
range_arr = np.arange(0, 10, 2)
```

11 . Generate evenly spaced numbers using linspace:

```
linspace_arr = np.linspace(0, 1, 5)
```

12 . Perform element - wise addition :

```
sum_arr = arr + 2
```

13 . Perform element - wise subtraction :

sub_arr = arr – 2

14 . Perform element - wise multiplication :

mul_arr = arr * 2

15 . Perform element - wise division :

div_arr = arr / 2

16 . Calculate the dot product of two arrays :

dot_product = np.dot(arr, arr)

17 . Transpose a matrix :

```
transpose_matrix = matrix.T
```

18 . Find the maximum element in an array :

```
max_val = arr.max()
```

19 . Find the minimum element in an array :

```
min_val = arr.min()
```

20 . Find the index of the maximum element :

```
max_index = arr.argmax()
```

21 . Find the index of the minimum element :

```
min_index = arr.argmin()
```

22 . Calculate the mean of an array :

```
mean_val = arr.mean()
```

23 . Calculate the median of an array :

```
median_val = np.median(arr)
```

24 . Calculate the standard deviation of an array :

```
std_dev = arr.std()
```

25 . Perform element - wise comparison :

```
bool_arr = arr > 3
```

26 . Slice and select elements from an array :

```
selected_elements = arr[1:4]
```

27 . Perform element - wise logical operations :

```
logical_and = np.logical_and(arr > 2, arr < 5)
```

28 . Find unique elements in an array :

```
unique_vals = np.unique(arr)
```

29 . Concatenate arrays horizontally :

```
hstacked = np.hstack((arr, arr))
```

30 . Concatenate arrays vertically :

```
vstacked = np.vstack((arr, arr))
```

31 . Create a random 1D array :

```
random_arr = np.random.rand(5)
```

32 . Create a random 2D array :

```
random_matrix = np.random.rand(3, 3)
```

33 . Create a random integer array :

```
random_ints = np.random.randint(1, 10, size=(3, 3))
```

34 . Reshape and flatten an array :

```
flattened = random_matrix.ravel()
```

35 . Sum elements along a specific axis :

```
sum_axis_0 = np.sum(matrix, axis=0)
```

36 . Find the index of specific elements :

```
indices = np.where(arr == 3)
```

37 . Sort an array :

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

38 . Reverse an array :

```
reversed_arr = np.flip(arr)
```

39 . Create a boolean mask :

```
mask = arr > 2
```

40 . Apply a mask to filter elements :

```
filtered_arr = arr[mask]
```

41 . Calculate element - wise square root :

```
sqrt_arr = np.sqrt(arr)
```

42 . Calculate element - wise exponential :

```
exp_arr = np.exp(arr)
```

43 . Calculate element - wise logarithm :

```
log_arr = np.log(arr)
```

44 . Calculate element - wise sine :

```
sin_arr = np.sin(arr)
```

45 . Calculate element - wise cosine :

cos_arr = np.cos(arr)

46 . Calculate element - wise absolute values :

abs_arr = np.abs(arr)

47 . Generate a diagonal matrix :

diag_matrix = np.diag([1, 2, 3])

48 . Calculate the eigenvalues and eigenvectors of a matrix :

eigenvalues, eigenvectors = np.linalg.eig(matrix)

49 . Calculate the matrix determinant :

```
determinant = np.linalg.det(matrix)
```

50 . Calculate the matrix inverse :

```
inverse_matrix = np.linalg.inv(matrix)
```

51 . Solve a system of linear equations :

```
solution = np.linalg.solve(matrix, np.array([6, 15]))
```

52 . Check if any elements are True in a boolean array :

```
any_true = np.any(arr > 3)
```

53 . Check if all elements are True in a boolean array :

```
all_true = np.all(arr > 0)
```

54 . Set operations with NumPy arrays :

```
set_intersection = np.intersect1d(arr1, arr2)
set_union = np.union1d(arr1, arr2)
set_difference = np.setdiff1d(arr1, arr2)
```

55 . Calculate element - wise absolute differences :

```
abs_diff = np.abs(arr1 - arr2)
```

56 . Calculate the correlation coefficient :

```
correlation_coeff = np.corrcoef(arr1, arr2)
```

57 . Reshape arrays using -1 for automatic dimension calculation :

```
reshaped_auto = arr.reshape(-1, 5)   # Reshape to 5 columns, calculate rows automatically .
```

58 . Calculate the cumulative sum of elements :

```
cumulative_sum = np.cumsum(arr)
```

59 . Calculate the cumulative product of elements :

```
cumulative_product = np.cumprod(arr)
```

60 . Find the indices that would sort an array :

```
sorted_indices = np.argsort(arr)
```

61 . Find the nearest value in an array :

```
nearest_value = arr[np.abs(arr - target).argmin()]
```

62 . Create a boolean mask for NaN values :

```
nan_mask = np.isnan(arr)
```

63 . Replace NaN values with a specific value :

```
arr[nan_mask] = replacement_value
```

64 . Calculate the 2D convolution of two matrices :

```
convolved_matrix = np.convolve(matrix1, matrix2, mode='full')
```

65 . Apply element - wise functions to an array using np.vectorize:

```
def my_function( x ):
    return x ** 2

vectorized_function = np.vectorize(my_function)
result = vectorized_function(arr)
```

66 . Calculate the Kronecker product of two arrays :

```
kron_product = np.kron(matrix1, matrix2)
```

67 . Calculate the Hadamard product of two arrays :

```
hadamard_product = np.multiply(matrix1, matrix2)
```

68 . Check if two arrays are element - wise equal :

```
are_equal = np.array_equal(arr1, arr2)
```

69 . Clip array values within a specified range :

```
clipped_arr = np.clip(arr, min_val, max_val)
```

70 . Calculate the element - wise reciprocal :

```
reciprocal_arr = 1  / arr
```

71 . Calculate the element - wise floor division :

```
floor_division_arr = np.floor_divide(arr, 2)
```

72 . Create a 3D NumPy array :

```
three_dimensional = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

73 . Calculate the mean along a specific axis :

```
mean_along_axis0 = np.mean(matrix, axis=0)
```

74 . Calculate the median along a specific axis :

```
median_along_axis1 = np.median(matrix, axis=1)
```

75 . Calculate the standard deviation along a specific axis :

```
std_dev_along_axis1 = np.std(matrix, axis=1)
```

76 . Calculate the sum along multiple axes :

```
sum_along_axes = np.sum(matrix, axis=(0, 1))
```

77 . Generate a random permutation of an array :

```
permuted_arr = np.random.permutation(arr)
```

78 . Compute the cross product of two arrays :

```
cross_product = np.cross(arr1, arr2)
```

79 . Apply a custom function to elements along an axis :

```
def custom_function( x ):
    return x.max() - x.min()

result = np.apply_along_axis(custom_function, axis=0, arr=matrix)
```

80 . Perform element - wise exponentiation :

```
exponentiated_arr = np.power(arr, 2)
```

81 . Calculate the cumulative minimum and maximum :

```
cumulative_min = np.minimum.accumulate(arr)
cumulative_max = np.maximum.accumulate(arr)
```

82 . Perform element - wise rounding :

```
rounded_arr = np.round(arr, decimals=2)
```

83 . Create a masked array :

```
masked_arr = np.ma.masked_where(arr < 2, arr)
```

84 . Find the unique values and their counts :

```
unique_values, counts = np.unique(arr, return_counts=True)
```

85 . Calculate the Kruskal - Wallis H - test for non - parametric analysis of variance :

```
from scipy.stats import kruskal

h_statistic, p_value = kruskal(arr1, arr2, arr3)
```

86 . Perform element - wise cube root :

```
cube_root_arr = np.cbrt(arr)
```

87 . Create a diagonal matrix with specified diagonals :

```
diagonal_matrix = np.diagflat([1, 2, 3], k=1)
```

88 . Calculate the element - wise absolute differences between two arrays :

```
absolute_diff = np.abs(arr1 - arr2)
```

89 . Compute the outer product of two vectors :

```
outer_product = np.outer(vector1, vector2)
```

90 . Calculate the percentile of elements in an array :

```
percentile_90 = np.percentile(arr, 90)
```

91 . Create a repeating pattern of an array :

```
repeated_pattern = np.tile(arr, 3)
```

92 . Calculate the Kronecker sum of two matrices :

```
kron_sum = np.kron(matrix1, np.ones_like(matrix2)) +
np.kron(np.ones_like(matrix1), matrix2)
```

93 . Calculate the cumulative product along a specific axis :

```
cumulative_product_axis0 = np.cumprod(matrix, axis=0)
```

94 . Calculate the mode of an array :

```
from scipy.stats import mode

mode_value = mode(arr).mode[0]
```

95 . Calculate the Poisson distribution :

```
from scipy.stats import poisson

poisson_arr = poisson.rvs(mu=2, size=10)
```

96 . Calculate the binomial distribution :

```
from scipy.stats import binom

binom_arr = binom.rvs(n=10, p=0.3, size=5)
```

97 . Calculate the mean absolute error (MAE) between two arrays :

```
mae = np.mean(np.abs(predicted_values - actual_values))
```

98 . Calculate the root mean squared error (RMSE) between two arrays :

```
rmse = np.sqrt(np.mean((predicted_values - actual_values) ** 2))
```

99 . Perform a matrix - vector multiplication :

```
result_vector = np.matmul(matrix, vector)
```

100 . Calculate the 1D discrete Fourier Transform (DFT):

```
dft = np.fft.fft(arr)
```

101 . Create a 3x3 matrix with random integers between 0 and 9 :

```
random_matrix = np.random.randint(0, 10, (3, 3))
print(random_matrix)
```

102 . Reshape a 1D array into a 2D array with automatic calculation of dimensions :

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

103 . Calculate the cross - correlation between two 1D arrays :

```
from scipy.signal import correlate

arr1 = np.array([1, 2, 1, 1])
arr2 = np.array([0, 1, 0.5])
correlation = correlate(arr1, arr2, mode='valid')
print(correlation)
```

104 . Calculate the cumulative sum along both rows and columns of a 2D array :

```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
cumulative_sum_rows = np.cumsum(matrix, axis=1)
cumulative_sum_columns = np.cumsum(matrix, axis=0)
print("Cumulative Sum (Rows):\n", cumulative_sum_rows)
print("Cumulative Sum (Columns):\n", cumulative_sum_columns)
```

105 . Create a boolean mask for values that meet multiple conditions :

```
arr = np.array([1, 2, 3, 4, 5, 6])
mask = (arr > 2) & (arr < 6)
filtered_arr = arr[mask]
print(filtered_arr)
```

106 . Calculate the moving average of a 1D array using convolution :

```
def  moving_average( arr, window_size ):
    kernel = np.ones(window_size) / window_size
    moving_avg = np.convolve(arr, kernel, mode='valid')
    return moving_avg

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
window_size = 3
result = moving_average(arr, window_size)
print(result)
```

107 . Calculate the matrix product of two 2D arrays :

```
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
matrix_product = np.matmul(matrix1, matrix2)
```

```
print(matrix_product)
```

108 . Calculate the pairwise Euclidean distances between rows in a 2D array :

```
from scipy.spatial import distance

matrix = np.array([[1, 2], [3, 4], [5, 6]])
distances = distance.cdist(matrix, matrix, 'euclidean')
print(distances)
```

109 . Create a mask for outliers in a dataset using the z - score :

```
def  find_outliers_zscore( data, threshold =2.0):
    z_scores = (data - np.mean(data)) / np.std(data)
    mask = np.abs(z_scores) > threshold
    return mask

data = np.array([1, 2, 2, 3, 4, 5, 100])
outlier_mask = find_outliers_zscore(data)
print(outlier_mask)
```

110 . Fit a polynomial regression model to data points using NumPy's polynomial functions :

```
import matplotlib.pyplot as plt

# Generate sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 7, 10, 15])

# Fit a polynomial regression model
coefficients = np.polyfit(x, y, 2)   # Fit a 2nd - degree polynomial
poly = np.poly1d(coefficients)

# Predict values
x_pred = np.linspace(0, 6, 100)
y_pred = poly(x_pred)

# Plot the data and the fitted curve
plt.scatter(x, y, label='Data')
plt.plot(x_pred, y_pred, label='Fitted Curve', color='red')
plt.legend()
plt.show()
```

111 . Calculate the element - wise natural logarithm of a 1D array and handle zero values :

```
def  safe_log( arr ):
    arr[arr == 0] = 1e-10   # Replace zeros with a small value to avoid
division by zero
    log_arr = np.log(arr)
    return log_arr

arr = np.array([1, 2, 0, 3, 0, 4])
log_arr = safe_log(arr)
print(log_arr)
```

112 . Create a heatmap using NumPy and Matplotlib :

```
import matplotlib.pyplot as plt

# Create a random 2D array
data = np.random.rand(5, 5)
```

```
# Create a heatmap
plt.imshow(data, cmap='viridis')
plt.colorbar()
plt.show()
```

113 . Calculate the mean of a 2D array along a specific axis, ignoring NaN values :

```
matrix = np.array([[1, 2, np.nan], [4, np.nan, 6], [7, 8, 9]])
mean_along_columns = np.nanmean(matrix, axis=0)
print(mean_along_columns)
```

114 . Simulate coin flips and calculate the probability of heads using random sampling :

```
n_flips = 1000
coin_flips = np.random.choice(['H', 'T'], size=n_flips)
n_heads = np.sum(coin_flips == 'H')
probability_heads = n_heads / n_flips
print("Probability of Heads:", probability_heads)
```

115 . Generate a 2D Gaussian distribution and create a contour plot :

```
import matplotlib.pyplot as plt

# Generate data
x, y = np.meshgrid(np.linspace(-3, 3, 100), np.linspace(-3, 3, 100))
z = np.exp(-(x**2  + y**2) / 2) / (2  * np.pi)

# Create a contour plot
plt.contourf(x, y, z, cmap='viridis')
plt.colorbar()
plt.title('2D Gaussian Distribution')
plt.show()
```

116 . Calculate the Pearson correlation coefficient between two 1D arrays :

```
from scipy.stats import pearsonr

arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([2, 3, 4, 5, 6])
correlation_coefficient, p_value = pearsonr(arr1, arr2)
print("Pearson Correlation Coefficient:", correlation_coefficient)
```

117 . Perform element - wise conditional assignment using np.where:

```
arr = np.array([1, 2, 3, 4, 5])
new_arr = np.where(arr > 3, 0, arr)
print(new_arr)
```

118 . Calculate the eigenvalues and eigenvectors of a real symmetric matrix :

```
from scipy.linalg import eig

matrix = np.array([[4, -2, 2], [-2, 3, -2], [2, -2, 2]])
eigenvalues, eigenvectors = eig(matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

119 . Calculate the 2D discrete Fourier Transform (DFT) and visualize the magnitude spectrum :

```
import matplotlib.pyplot as plt

# Create a 2D sinusoidal pattern
x, y = np.meshgrid(np.linspace(0, 1, 100), np.linspace(0, 1, 100))
frequencies = 10   # Number of cycles
z = np.sin(2  * np.pi * frequencies * (x + y))

# Compute the 2D DFT
dft = np.fft.fft2(z)
magnitude_spectrum = np.abs(dft)

# Visualize the magnitude spectrum
plt.imshow(np.log(1  + magnitude_spectrum), cmap='viridis')
plt.colorbar()
plt.title('Magnitude Spectrum (log-scale)')
plt.show()
```

120 . Fit a logistic regression model to a dataset using NumPy and gradient descent :

```
# Generate sample data
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]])
y = np.array([0, 0, 1, 1, 1])

# Initialize model parameters
theta = np.zeros(X.shape[1])

# Define sigmoid function
def  sigmoid( x ):
    return  1  / (1  + np.exp(-x))

# Gradient descent
learning_rate = 0.1
num_iterations = 1000

for _ in  range(num_iterations):
    predictions = sigmoid(np.dot(X, theta))
    gradient = np.dot(X.T, (predictions - y)) / len(y)
    theta -= learning_rate * gradient

print("Optimal Theta:", theta)
```

121 . Calculate the element - wise absolute differences between two 2D arrays and find the maximum difference :

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[3, 1], [4, 6]])
absolute_diff = np.abs(arr1 - arr2)
max_diff = np.max(absolute_diff)
print("Absolute Differences:\n", absolute_diff)
print("Maximum Difference:", max_diff)
```

122 . Implement a simple k - means clustering algorithm from scratch using NumPy :

```
# Generate random data points
data = np.random.rand(100, 2)

# Define the number of clusters
k = 3

# Initialize cluster centroids randomly
centroids = data[np.random.choice(len(data), k, replace=False)]

# Perform k - means clustering
num_iterations = 100
for _ in  range(num_iterations):
    # Assign each point to the nearest centroid
    distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)

    # Update centroids
    for i in  range(k):
        centroids[i] = np.mean(data[labels == i], axis=0)

# Visualize the clustered data points and centroids
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=100)
plt.title('K-Means Clustering')
plt.show()
```

123 . Calculate the element - wise mean squared error (MSE) between two images :

```
from scipy.ndimage import imread

# Load two images ( grayscale )
image1 = imread('image1.png', mode='L')
image2 = imread('image2.png', mode='L')

# Calculate MSE
mse = np.mean((image1 - image2) ** 2)
print("Mean Squared Error:", mse)
```

124 . Create a 2D array with values from a custom function applied element - wise :

```
def custom_function( x, y ):
    return x + y * 2

x_values = np.linspace(1, 5, 5)
y_values = np.linspace(6, 10, 5)
result = np.fromfunction(np.vectorize(custom_function), (5, 5), x=x_values, y=y_values)
print(result)
```

125 . Implement the Run - Length Encoding (RLE) algorithm to compress a 1D binary array :

```
def run_length_encode( arr ):
    encoded = []
    current_run = arr[0]
    run_length = 1

    for i in range(1, len(arr)):
        if arr[i] == current_run:
            run_length += 1
        else:
            encoded.append((current_run, run_length))
            current_run = arr[i]
            run_length = 1

    encoded.append((current_run, run_length))
    return encoded

binary_array = np.array([1, 1, 0, 0, 1, 1, 1, 0, 0, 0])
encoded_result = run_length_encode(binary_array)
print(encoded_result)
```

126 . Use NumPy to calculate the integral of a function over a specified range :

```
from scipy.integrate import quad

# Define the function to integrate
def my_function( x ):
    return x ** 2

# Integrate the function from 0 to 1
result, _ = quad(my_function, 0, 1)
print("Integral:", result)
```

127 . Calculate the element - wise Hamming distance between two 1D arrays :

```
def hamming_distance( arr1, arr2 ):
    return np.sum(arr1 != arr2)

binary1 = np.array([0, 1, 1, 0, 1])
binary2 = np.array([1, 1, 0, 0, 1])
distance = hamming_distance(binary1, binary2)
print("Hamming Distance:", distance)
```

128 . Calculate the outer product of two vectors and visualize it as a matrix :

```
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
outer_product = np.outer(vector1, vector2)
print("Outer Product:\n", outer_product)
```

129 . Generate a random permutation of an array using Fisher - Yates shuffle :

```
def fisher_yates_shuffle( arr ):
    n = len(arr)
    for i in range(n - 1, 0, -1):
        j = np.random.randint(0, i + 1)
        arr[i], arr[j] = arr[j], arr[i]

original_arr = np.array([1, 2, 3, 4, 5])
shuffled_arr = original_arr.copy()
fisher_yates_shuffle(shuffled_arr)
print("Original Array:", original_arr)
print("Shuffled Array:", shuffled_arr)
```

130 . Calculate the 2D convolution of an image and a kernel using NumPy :

```
from scipy.signal import convolve2d
import matplotlib.pyplot as plt

# Create a simple image
image = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# Create a kernel
kernel = np.array([[0, 1, 0],
                   [1, -4, 1],
                   [0, 1, 0]])

# Perform convolution
result = convolve2d(image, kernel, mode='same')

# Visualize the original image and the result
plt.subplot(121)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot(122)
plt.imshow(result, cmap='gray')
plt.title('Convolved Image')
plt.show()
```

131 . Create a random symmetric positive - definite matrix and perform a Cholesky decomposition :

```
# Generate a random symmetric matrix
A = np.random.rand(3, 3)
symmetric_matrix = A @ A.T

# Perform Cholesky decomposition
L = np.linalg.cholesky(symmetric_matrix)

# Verify that L is lower triangular and LL^T equals the original matrix
print("Cholesky Decomposition (L):\n", L)
print("Reconstructed Matrix (LL^T):\n", L @ L.T)
```

132 . Create a simple feedforward neural network with random weights and perform forward pass :

```
# Define the input data
input_data = np.array([0.5, 0.2, 0.7])

# Define the network architecture  ( weights and biases )
input_size = 3
hidden_size = 4
output_size = 2

weights_input_hidden = np.random.rand(input_size, hidden_size)
biases_hidden = np.random.rand(hidden_size)

weights_hidden_output = np.random.rand(hidden_size, output_size)
biases_output = np.random.rand(output_size)

# Perform forward pass
hidden_layer_input = np.dot(input_data, weights_input_hidden) +
biases_hidden
hidden_layer_output = 1  / (1  + np.exp(-hidden_layer_input))

output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)
+ biases_output
output_layer_output = 1  / (1  + np.exp(-output_layer_input))

print("Output Layer Output:", output_layer_output)
```

133 . Calculate the element - wise outer product of a 1D array with itself :

```
arr = np.array([1, 2, 3, 4])
outer_product = np.outer(arr, arr)
print("Outer Product:\n", outer_product)
```

134 . Use NumPy to solve a system of linear equations with multiple variables :

```
# Define the coefficient matrix ( A ) and the right - hand side ( b )
A = np.array([[2, 1, -1],
              [1, 3, 2],
              [1, 0, 3]])
b = np.array([8, 13, 6])

# Solve for the unknowns ( x )
x = np.linalg.solve(A, b)
print("Solution (x):", x)
```

135 . Generate a random symmetric matrix and find its eigenvalues and eigenvectors :

```
# Generate a random symmetric matrix
symmetric_matrix = np.random.rand(4, 4)
symmetric_matrix = (symmetric_matrix + symmetric_matrix.T) / 2   #
Ensure symmetry

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(symmetric_matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

136 . Calculate the element - wise reciprocal of a 1D array, handling zero values :

```
def  reciprocal_with_zero_handling( arr ):
    reciprocal_arr = np.where(arr != 0, 1  / arr, np.inf)
    return reciprocal_arr

arr = np.array([1, 2, 0, 3, 0, 4])
reciprocal_arr = reciprocal_with_zero_handling(arr)
print(reciprocal_arr)
```

137 . Create a simple 3D array and calculate the mean along specified axes :

```
three_dimensional = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
mean_along_axis0 = np.mean(three_dimensional, axis=0)
mean_along_axis1 = np.mean(three_dimensional, axis=1)
mean_along_axis2 = np.mean(three_dimensional, axis=2)
print("Mean along Axis 0:\n", mean_along_axis0)
print("Mean along Axis 1:\n", mean_along_axis1)
print("Mean along Axis 2:\n", mean_along_axis2)
```

138 . Create a boolean mask for values within a specified range in a 1D array :

```
arr = np.array([1, 2, 3, 4, 5, 6])
lower_bound = 2
upper_bound = 5
mask = (arr >= lower_bound) & (arr <= upper_bound)
filtered_arr = arr[mask]
print(filtered_arr)
```

139 . Calculate the element - wise square root of a 2D array and handle negative values :

```
def  safe_sqrt( arr ):
    arr[arr < 0] = 0   # Replace negative values with 0
    sqrt_arr = np.sqrt(arr)
    return sqrt_arr

matrix = np.array([[1, 4], [-1, 9]])
sqrt_matrix = safe_sqrt(matrix)
print(sqrt_matrix)
```

140 . Create a 2D array with a specific pattern using NumPy's meshgrid :

```
x = np.linspace(-1, 1, 5)
y = np.linspace(-1, 1, 5)
x_grid, y_grid = np.meshgrid(x, y)
result = x_grid ** 2  + y_grid ** 2
print(result)
```

141 . Calculate the determinant of a 2D matrix and check if it's invertible :

```
matrix = np.array([[2, 1], [1, 3]])
determinant = np.linalg.det(matrix)
is_invertible = determinant != 0
print("Determinant:", determinant)
print("Is Invertible:", is_invertible)
```

142 . Implement a simple 2D convolution operation with a custom kernel :

```
def custom_convolution( image, kernel ):
    output = np.zeros_like(image)
    k_height, k_width = kernel.shape
    i_height, i_width = image.shape

    for i in range(i_height - k_height + 1):
        for j in range(i_width - k_width + 1):
            output[i, j] = np.sum(image[i:i + k_height, j:j + k_width] *
kernel)

    return output

# Create a simple image and kernel
image = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])

# Perform convolution
convolved_image = custom_convolution(image, kernel)
print("Convolved Image:\n", convolved_image)
```

143 . Calculate the element - wise reciprocal of a 2D array and handle zero values :

```
def reciprocal_with_zero_handling( arr ):
    reciprocal_arr = np.where(arr != 0, 1 / arr, np.inf)
    return reciprocal_arr

matrix = np.array([[1, 2], [0, 3], [0, 0]])
reciprocal_matrix = reciprocal_with_zero_handling(matrix)
print(reciprocal_matrix)
```

144 . Create a masked array to handle invalid values in a 1D array :

```
arr = np.array([1, 2, -999, 4, -999, 6])
mask = arr == -999
masked_arr = np.ma.masked_where(mask, arr)
print(masked_arr)
```

145 . Find the unique values in a 2D array along each column and count their occurrences :

```
matrix = np.array([[1, 2, 3], [2, 3, 4], [3, 4, 5], [2, 3, 4]])
unique_values = [np.unique(matrix[:, i]) for i in  range(matrix.shape[1])]
value_counts = [np.bincount(matrix[:, i]) for i in  range(matrix.shape[1])]
print("Unique Values for Each Column:", unique_values)
print("Value Counts for Each Column:", value_counts)
```

146 . Implement a 2D convolution operation with zero padding :

```
def  convolution_with_padding( image, kernel ):
    k_height, k_width = kernel.shape
    i_height, i_width = image.shape
    padding_height = k_height // 2
    padding_width = k_width // 2

    padded_image = np.pad(image, ((padding_height, padding_height),
(padding_width, padding_width)), mode='constant')
    output = np.zeros_like(image)

    for i in  range(i_height):
        for j in  range(i_width):
            output[i, j] = np.sum(padded_image[i:i + k_height, j:j + k_width]
* kernel)

    return output

# Create a simple image and kernel
image = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])

# Perform convolution with padding
convolved_image = convolution_with_padding(image, kernel)
```

```
print("Convolved Image with Padding:\n", convolved_image)
```

147 . Calculate the element - wise cosine similarity between rows in a 2D array :

```
from scipy.spatial.distance import cosine

matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
cosine_similarity = np.zeros((matrix.shape[0], matrix.shape[0]))

for i in  range(matrix.shape[0]):
    for j in  range(i, matrix.shape[0]):
        similarity = 1  - cosine(matrix[i], matrix[j])
        cosine_similarity[i, j] = similarity
        cosine_similarity[j, i] = similarity

print("Cosine Similarity Matrix:\n", cosine_similarity)
```

148 . Generate a random permutation of integers within a specified range :

```
n = 10
permutation = np.random.permutation(np.arange(1, n + 1))
print("Random Permutation:", permutation)
```

149 . Calculate the element - wise reciprocal of a 3D array and handle zero values :

```
def reciprocal_with_zero_handling( arr ):
    arr[arr == 0] = 1e-10   # Replace zeros with a small value to avoid
division by zero
    reciprocal_arr = 1  / arr
    return reciprocal_arr

tensor = np.array([[[1, 2], [3, 0]], [[0, 5], [6, 7]]])
reciprocal_tensor = reciprocal_with_zero_handling(tensor)
print(reciprocal_tensor)
```

150 . Create a mask for values within a specified range in a 2D array and calculate their sum :

```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
min_val = 3
max_val = 7
mask = (matrix >= min_val) & (matrix <= max_val)
sum_of_masked_values = np.sum(matrix[mask])
print("Sum of Masked Values:", sum_of_masked_values)
```

These examples demonstrate a wide range of capabilities and applications of NumPy in Python, including matrix operations, image processing, statistical analysis, and more . NumPy is a versatile library that is essential for scientific computing and data manipulation .
