NumPy: Welcome to the World of Matrices

In [1]:
```
import numpy as np
```

Check out the Detailed Article in Medium - https://medium.com/@tejag311/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84 (https://medium.com/@tejag311/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84)

Table of Contents

1-Numpy Array Basics

Creating Numpy Arrays

In [2]:
```python
# Creating an integer array with explicit dtype, which is not necessary.
int_array = np.array([1, 2, 3], dtype=np.int32)
int_array
```

Out[2]:
```
array([1, 2, 3])
```

In [3]:
```python
# Create an 2D array
original_array = np.array([[1, 2, 3],
                           [4, 5, 6]])
original_array
```

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

In [4]:
```python
# Creating a 3D array (Tensor) # Dimension: 3 , Shape: (2, 2, 2)
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
arr_3d
```

Out[4]:
```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

In [5]:
```python
# Create an array of 10 equally spaced values from 0 to 1
linspace_arr = np.linspace(0, 1, 10)
linspace_arr
```

Out[5]:
```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

In [6]:
```python
# Create an array of 5 values spaced logarithmically from 1 to 100
logspace_arr = np.logspace(0, 2, 5)
logspace_arr
```

Out[6]:
```
array([  1.        ,   3.16227766,  10.        ,  31.6227766 ,
       100.        ])
```

In [7]:
```python
# Create an array of values from 0 to 9 with a step size of 2
arange_arr = np.arange(0, 10, 2)
arange_arr
```

Out[7]:
```
array([0, 2, 4, 6, 8])
```

In [8]:
```python
# Create a 3x3 array filled with zeros
zeros_arr = np.zeros((3, 3))
zeros_arr
```

Out[8]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [9]:
```python
# Create a 2x4 array filled with ones
ones_arr = np.ones((2, 4))
ones_arr
```

Out[9]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [10]:
```python
# Create a new array filled with zeros,
# matching the shape and data type of the original array
zeros_array = np.zeros_like(original_array)
zeros_array
```

Out[10]:
```
array([[0, 0, 0],
       [0, 0, 0]])
```

In [11]:
```
# Create a new array filled with ones,
# matching the shape and data type of the original array
ones_array = np.ones_like(original_array)
ones_array
```

Out[11]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

2-Array Inspection

2.1 Array Dimension and Shapes

In [12]:
```
# Creating a 1D array (Vector)
arr_1d = np.array([1, 2, 3])
# Dimesion: 1 , Shape: (3,), Size: 3
print(f"Dimension: {arr_1d.ndim}, Shape: {arr_1d.shape}, size: {arr_1d.size}")
```

```
Dimension: 1, Shape: (3,), size: 3
```

2.2 Array Indexing and Slicing

In [13]:
```
# Creating a NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Accessing individual elements
first_element = arr[0]  # Access the first element (10)
print(f'First element - {first_element}')

third_element = arr[2]  # Access the third element (30)
print(f'Third element - {third_element}')

# Accessing elements using negative indices
last_element = arr[-1]  # Access the last element (50)
print(f'Last element - {last_element}')
```

```
First element - 10
Third element - 30
Last element - 50
```

In [14]:
```
# Creating a 2D NumPy array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Slicing along rows and columns
sliced_array = arr_2d[0,1]  # Element at row 0, column 1 (value: 2)
sliced_array
```

Out[14]:
```
2
```

In [15]:
```
# Slicing the array to create a new array
sliced_array = arr[1:4]  # Slice from index 1 to 3 (exclusive) [20,30,40]
sliced_array
```

Out[15]:
```
array([20, 30, 40])
```

In [16]:
```
# Slicing with a step of 2
sliced_array = arr[0::2]  # Start at index 0, step by 2 [10,30,50]
sliced_array
```

Out[16]:
```
array([10, 30, 50])
```

In [17]:
```
# Slicing with negative index
second_to_last = arr[-2::]  # Access the last two elements [40,50]
second_to_last
```

Out[17]:
```
array([40, 50])
```

In [18]:
```
# Conditional slicing: Select elements greater than 30
sliced_array = arr[arr > 30]  # Result: [40, 50]
sliced_array
```

Out[18]:
```
array([40, 50])
```

In [19]:
```
# Creating a 2D NumPy array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Slicing along rows and columns
sliced_array = arr_2d[1:3, 0:2] # Slice a 2x2 subarray: [[4, 5], [7, 8]]
sliced_array
```

Out[19]:
```
array([[4, 5],
       [7, 8]])
```

3-Array Operations

3.1 Element-wise Operations

In [20]:
```
# Creating 1D NumPy arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
scalar = 2

# Addition
result_add = arr1 + arr2  # [5, 7, 9]
result_add
```

Out[20]:
```
array([5, 7, 9])
```

In [21]:
```
# Multiplication, Similarly subraction and division as well.
result_mul = arr1 * arr2  # [4, 10, 18]
result_mul
```

Out[21]:
```
array([ 4, 10, 18])
```

In [22]:
```
# Creating 2D NumPy arrays
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])

# Multiplication (element-wise, not matrix multiplication)
result_mul = matrix1 * matrix2  # [[5, 12], [21, 32]]
result_mul
```

Out[22]:
```
array([[ 5, 12],
       [21, 32]])
```

In [23]:
```
# Actual Matrix Multiplication using np.dot
matrix_multiplication = np.dot(matrix1,matrix2)
matrix_multiplication
```

Out[23]:
```
array([[19, 22],
       [43, 50]])
```

In [24]:
```
# Broadcasting: Multiply array by a scalar
result = arr1 * scalar  # [2, 4, 6]
result
```

Out[24]:
```
array([2, 4, 6])
```

3.2 Append and Delete

In [25]:
```
# Create an array
original_array = np.array([1, 2, 3])

# Append elements in-place
original_array = np.append(original_array, [4, 5, 6])
original_array
```

Out[25]:
```
array([1, 2, 3, 4, 5, 6])
```

In [26]:
```
# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Remove the item at index 2 (value 3)
new_arr = np.delete(arr, 2)
new_arr
```

Out[26]:
```
array([1, 2, 4, 5])
```

```
In [27]: # Create a 2D NumPy array
         arr = np.array([[1, 2, 3], [4,5, 6], [7, 8, 9]])

         # Remove the second row (index 1)
         new_arr = np.delete(arr, 1, axis=0)
         new_arr

Out[27]: array([[1, 2, 3],
                [7, 8, 9]])
```

3.3 Aggregation Functions and ufuncs

```
In [28]: # Creating a NumPy array
         arr = np.array([1, 2, 3, 4, 5])

         # Aggregation functions
         mean_value = np.mean(arr)  # Mean: 3.0
         print(mean_value)

         3.0
```

```
In [29]: median_value = np.median(arr) # Median: 3.0
         variance = np.var(arr) # Variance 2.0
         standard_deviation = np.std(arr) # std: 1.414

         sum_value = np.sum(arr)    # Sum: 15
         min_value = np.min(arr)    # Minimum: 1
         max_value = np.max(arr)    # Maximum: 5
```

```
In [30]: # Universal functions
         sqrt_arr = np.sqrt(arr) # Square Root
         print(f'square root - {sqrt_arr}')
         exp_arr = np.exp(arr) # Exponential
         print(f'exponential array - {exp_arr}')

         square root - [1.         1.41421356 1.73205081 2.         2.23606798]
         exponential array - [  2.71828183   7.3890561   20.08553692  54.59815003 148.413
         1591  ]
```

3.4 Reshaping Arrays

```
In [31]: # Creating 2D array
         arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

         # Here as we are passing only 6, it will conver the 2d array to a 1d array
         # with 6 elements, you cannot pass anything other than 6,
         # as it doesn't match the original array!
         arr_1d = arr_2d.reshape(6)
         arr_1d

Out[31]: array([1, 2, 3, 4, 5, 6])
```

In [32]:
```python
# Creating 1D array
arr_1d = np.array([1, 2, 3, 4, 5, 6])

# converting 1D array to 2D
arr_2d = arr_1d.reshape(2, 3)
arr_2d
```

Out[32]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

understanding how to use -1: You can use -1 as a placeholder in any one dimension of the new shape, and NumPy will automatically calculate the size for that dimension.

In [33]:
```python
from skimage import data
# Load a sample grayscale image
image = data.coins()

# Original shape of the image
print("Original Image Shape:", image.shape) # Original Image Shape: (303, 384)

# So, if you want to convert it to 1D, you have to pass 116352 (303*384)
# Instead, if you don't want to calculate that and let numpy deal with it,
# IN such cases, you can just pass -1, and it will calculate 116352
reshaped_image = image.reshape(-1)

print("Reshapped array:", reshaped_image.shape)
```

```
Original Image Shape: (303, 384)
Reshapped array: (116352,)
```

In [34]:
```python
# Creating a 1D array with 12 elements
arr = np.arange(12)
print("original array shape:",arr.shape)
# Reshaping into a 2D array with an unknown number of columns (-1)
reshaped_arr = arr.reshape(4, -1)
print("Reshaped array shape:",reshaped_arr.shape)
```

```
original array shape: (12,)
Reshaped array shape: (4, 3)
```

- so you can see that the 3 is calculate automatically by just giving -1

4-Working with Numpy Arrays

4.1 Combining Arrays

In [35]:
```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Concatenate along the 0-axis (rows)
combined = np.concatenate((arr1, arr2)) # Result: [1, 2, 3, 4, 5, 6]
combined
```

Out[35]:
```
array([1, 2, 3, 4, 5, 6])
```

In [36]:
```
# Vertical stacking
vertical_stack = np.vstack((arr1, arr2)) # Result: [[1, 2, 3], [4, 5, 6]]
vertical_stack
```

Out[36]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [37]:
```
# Horizontal stacking
horizontal_stack = np.hstack((arr1, arr2)) # Result: [1, 2, 3, 4, 5, 6]
horizontal_stack
```

Out[37]:
```
array([1, 2, 3, 4, 5, 6])
```

4.2 Splitting Arrays

In [38]:
```
arr = np.array([1, 2, 3, 4, 5, 6])

# Split into three equal parts
split_arr = np.split(arr, 3)
split_arr
# Result: [array([1, 2]), array([3, 4]), array([5, 6])].
```

Out[38]:
```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

4.3 Alias vs. View vs. Copy of Arrays

- Alias: An alias refers to multiple variables that all point to the same underlying NumPy array object. They share the same data in memory. Changes in alias array will affect the original array.
- View: The .view() method creates a new array object that looks at the same data as the original array but does not share the same identity. It provides a way to view the data differently or with different data types, but it still operates on the same underlying data.
- Copy: A copy is a completely independent duplicate of a NumPy array. It has its own data in memory, and changes made to the copy will not affect the original array, and vice versa.

In [39]:
```
original_arr = np.array([1, 2, 3])

# alias of original array
alias_arr = original_arr

# chaing a value in alias array
alias_arr[0]=10

# you can observe that it will also change the original array
original_arr
```

Out[39]:
```
array([10,  2,  3])
```

In [40]:
```
original_arr = np.array([1, 2, 3])
# Changes to view_arr will affect the original array
view_arr = original_arr.view()

# Modify an element in the view
view_arr[0] = 99

# Check the original array
print(original_arr)
```

```
[99  2  3]
```

In [41]:
```
original_arr = np.array([1, 2, 3])
# Changes to copy_arr won't affect the original array
copy_arr = original_arr.copy()

copy_arr[0] = 100

# Copy doesn't change the original array
print(original_arr)
```

```
[1 2 3]
```

4.4 Sorting Numpy Arrays

In [42]:
```
data = np.array([3, 1, 5, 2, 4])
sorted_data = np.sort(data)  # Ascending order
print("Ascending sort", sorted_data)

reverse_sorted_data = np.sort(data)[::-1]  # Descending order
print("Descending sort", reverse_sorted_data)

# Returns Indices that would sort the array.
sorted_indices = np.argsort(data)
print("Sorted Indices", sorted_indices)
```

```
Ascending sort [1 2 3 4 5]
Descending sort [5 4 3 2 1]
Sorted Indices [1 3 0 4 2]
```

5-Numpy for Data Cleaning

5.1 Identify Missing Values

NumPy provides functions to check for missing values in a numeric array, represented as NaN (Not a Number).

In [43]:
```
# Create a NumPy array with missing values
data = np.array([1, 2, np.nan, 4, np.nan, 6])

# Check for missing values
has_missing = np.isnan(data)
print(has_missing)
```

```
[False False  True False  True False]
```

5.2 Removing rows or columns with Missing Values

We can use np.isnan to get a boolean matrix with True for the indices where there is a missing value. And when we pass it to np.any, it will return a 1D array with True for the index where any row item is True. And finally we ~ (not), and pass the boolean to the original Matrix, which will remove the rows with missing values.

In [44]:
```
# Create a 2D array with missing values
data = np.array([[1, 2, 3], [4, np.nan, 6], [7, 8, 9]])

# Remove rows with any missing values
cleaned_data = data[~np.any(np.isnan(data), axis=1)]
print(cleaned_data) # Result: [[1,2,3],[7,8,9]]
```

```
[[1. 2. 3.]
 [7. 8. 9.]]
```

6-Numpy for Statistical Analysis

6.1 Data Transformation

Data transformation involves modifying data to meet specific requirements or assumptions. Numpy doesn’t have these features directly, but we can utilize the existing features to perform these.

In [45]:
```python
# Data Centering
data = np.array([10, 20, 30, 40, 50])
mean = np.mean(data)
centered_data = data - mean
print('Centered data = ',centered_data)

# Standardization
std_dev = np.std(data)
standardized_data = (data - mean) / std_dev
print("standardized data = ",standardized_data)

# Log Transformation
log_transformed_data = np.log(data)
print("log_transformed_data = ",log_transformed_data)
```

```
Centered data =  [-20. -10.   0.  10.  20.]
standardized data =  [-1.41421356 -0.70710678  0.          0.70710678  1.4142135
6]
log_transformed_data =  [2.30258509 2.99573227 3.40119738 3.68887945 3.91202301]
```

6.2 Random Sampling and Generation

Sampling

- Simple Random Sampling: Select a random sample of a specified size from a dataset. When sampling without replacement, each item selected is not returned to the population.
- Bootstrap Sampling: Bootstrap sampling involves sampling with replacement to create multiple datasets. This is often used for estimating statistics' variability. # Simple Random Sampling W

In [46]:
```python
# Simple Random Sampling Without replacement
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
random_samples = np.random.choice(data, size=5, replace=False)
random_samples
```

Out[46]:
```
array([2, 8, 1, 7, 6])
```

In [47]:
```python
# Bootstrap Sampling
num_samples = 1000
bootstrap_samples = np.random.choice(data, size=(num_samples, len(data)), replac
e=True)
bootstrap_samples
```

Out[47]:
```
array([[ 9,  7,  4, ...,  4,  6,  8],
       [ 9,  8,  4, ...,  6,  5,  7],
       [ 7,  3, 10, ...,  8,  7,  6],
       ...,
       [ 6,  3,  8, ...,  5,  3,  6],
       [ 3, 10,  8, ...,  3,  8,  7],
       [ 8,  3, 10, ...,  1,  8,  3]])
```

Generation

In [48]:
```
np.random.randint(0,100)
```
Out[48]:
```
99
```

In [49]:
```
# Generates 5 random values from a standard normal distribution
mean = 0
std_dev = 1
normal_values = np.random.normal(mean, std_dev, 5)
print(normal_values)
```
```
[-0.20126629 -1.24514517 -0.37233003  1.64812603  1.94393548]
```

In [50]:
```
# Simulates 5 sets of 10 trials with a success probability of 0.5
n_trials = 10
probability = 0.5
binomial_values = np.random.binomial(n_trials, probability, 5)
print(binomial_values)
```
```
[4 2 5 9 5]
```

In [51]:
```
# Generates 5 random values following a Poisson distribution with a rate of 2.5
rate = 2.5
poisson_values = np.random.poisson(rate, 5)
print(poisson_values)
```
```
[4 1 3 8 2]
```

In [52]:
```
 # Generates 5 random values following an exponential distribution with a scale parameter of 0.5
scale_parameter = 0.5
exponential_values = np.random.exponential(scale_parameter, 5)
print(exponential_values)
```
```
[0.20435827 0.43703618 0.631526   0.27823771 0.6482197 ]
```

In [53]:
```
# Generates 5 random values following a log-normal distribution
mean_of_log = 0
std_dev_of_log = 0.5
lognormal_values = np.random.lognormal(mean_of_log, std_dev_of_log, 5)
print(lognormal_values)
```
```
[1.08121668 1.01669187 1.30597122 0.81159366 2.37190169]
```

In [54]:
```
# Simulates 5 sets of 10 multinomial trials with the given probabilities
n_trials = 10
probabilities = [0.2, 0.3, 0.5]  # Probabilities of each outcome
multinomial_values = np.random.multinomial(n_trials, probabilities, 5)
print(multinomial_values)
```

```
[[4 3 3]
 [1 0 9]
 [4 4 2]
 [2 3 5]
 [3 3 4]]
```

7-Numpy for Linear Algebra

7.1 Complex Matrix Operations

We have already seen Creating vectors, matrices, and the amazing matrix operations we can do with numpy. Now, Let’s see even complex matrix operations.

In [55]:
```
A = np.array([[1, 2], [3, 4]])

# Calculate the inverse of A
A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

In [56]:
```
A = np.array([[2, -1], [1, 1]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
print("eigenvalues:",eigenvalues)
print("eigenvectors:",eigenvectors)
```

```
eigenvalues: [1.5+0.8660254j 1.5-0.8660254j]
eigenvectors: [[0.35355339+0.61237244j 0.35355339-0.61237244j]
 [0.70710678+0.j         0.70710678-0.j        ]]
```

In [57]:
```
A = np.array([[1, 2], [3, 4], [5, 6]])

# Compute the  Singular Value Decomposition (SVD)
U, S, VT = np.linalg.svd(A)
```

7.2 Solve Linear Equations

Yes, You can even solve linear equations with numpy features. Solve systems of linear equations using np.linalg.solve()

In [58]:
```
A = np.array([[2, 3], [4, 5]])
b = np.array([6, 7])

# Solve Ax = b for x
x = np.linalg.solve(A, b)
print(x)
```

```
[-4.5  5. ]
```

8-Advanced Numpy Techniques

8.1 Masked Arrays

Masked arrays in NumPy allow you to work with data where certain elements are invalid or missing. A mask is a Boolean array that indicates which elements should be considered valid and which should be masked (invalid or missing).

Masked arrays enable you to perform operations on valid data while ignoring the masked elements.

In [59]:
```
import numpy.ma as ma

# Temperature dataset with missing values (-999 represents missing values)
temperatures = np.array([22.5, 23.0, -999, 24.5, -999, 26.0, 27.2, -999, 28.5])

# Calculate the mean temperature without handling missing values
mean_temperature = np.mean(temperatures)

# Print the result = -316.14
print("Mean Temperature (without handling missing values):", mean_temperature)


# Create a mask for missing values (-999)
mask = (temperatures == -999)

# Create a masked array
masked_temperatures = ma.masked_array(temperatures, mask=mask)

# Calculate the mean temperature (excluding missing values)
mean_temperature = ma.mean(masked_temperatures)

# Print the result = 25.28
print("Mean Temperature (excluding missing values):", mean_temperature)
```

```
Mean Temperature (without handling missing values): -316.14444444444445
Mean Temperature (excluding missing values): 25.283333333333333
```

8.2 Structured Arrays

Structured arrays allow you to work with heterogeneous data, similar to a table with named columns. Each element of a structured array can have different data types. Create your datatypes by using np.dtype and add the column name and datatype as a tuple. Then you can pass it to your array.

In [60]:
```
# Define data types for fields
dt = np.dtype([('name', 'S20'), ('age', int), ('salary', float)])

# Create a structured array
employees = np.array([('Alice', 30, 50000.0), ('Bob', 25, 60000.0)], dtype=dt)

# Access the 'name' field of the first employee
print(employees['name'][0])

# Access the 'age' field of all employees
print(employees['age'])
```

```
b'Alice'
[30 25]
```

Conclusion

In this NumPy guide, we've covered essential aspects and advanced techniques for data science and numerical computing. Remember, NumPy is a vast library with endless possibilities. What we have seen is still basic and we can do even a lot more, explore further to unlock its full potential and elevate your data-driven solutions.