# NumPy Overview

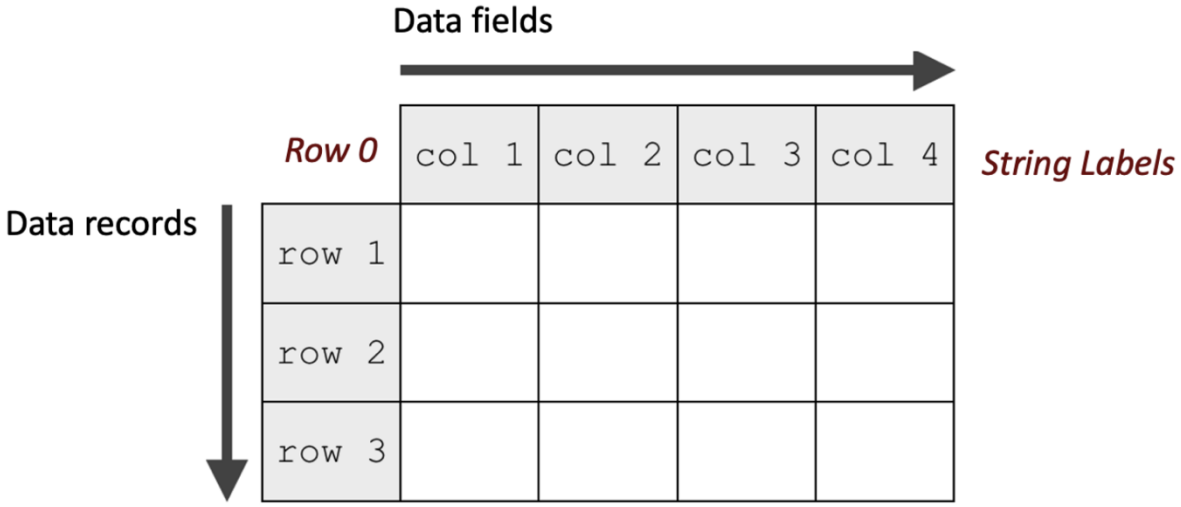
**Courses: Machine Learning in Nutshell (15-288), AI & ML I (07-280)**

**By: Mohammed Yusuf Ansari, PhD**

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

# Machine Learning Data

- Nature of Machine Learning Data:
  - Data is typically stored and processed in **tabular form**
  - Tabular data is naturally represented as a **matrix** of values
  - Each row corresponds to a **single data example**
  - Each column corresponds to a specific **property/attribute/feature** of the data



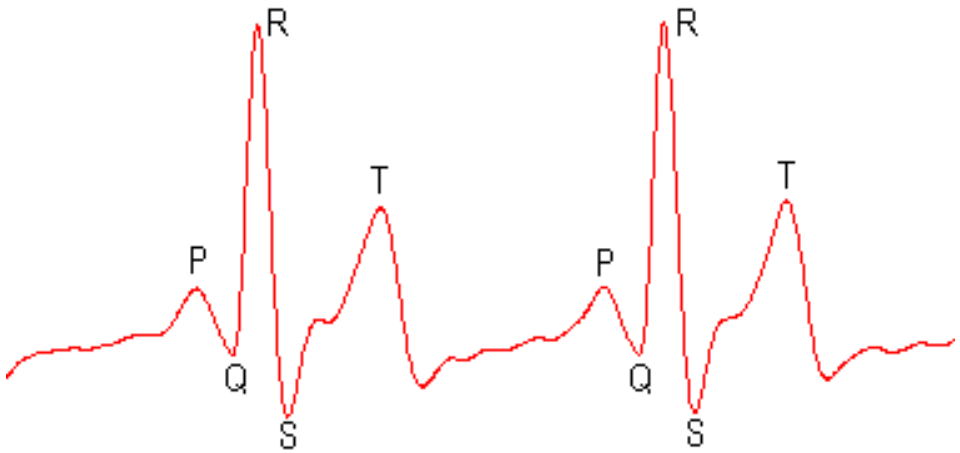

| index | age | sex | bmi | children | smoker | region | charges |
|---|---|---|---|---|---|---|---|
| 0 | 19 | female | 27.9 | 0 | yes | southwest | 16884.924 |
| 1 | 18 | male | 33.77 | 1 | no | southeast | 1725.5523 |
| 2 | 28 | male | 33 | 3 | no | southeast | 4449.462 |
| 3 | 33 | male | 22.705 | 0 | no | northwest | 21984.4706 |
| 4 | 32 | male | 28.88 | 0 | no | northwest | 3866.8552 |
| 5 | 31 | female | 25.74 | 0 | no | southeast | 3756.6216 |
| 6 | 46 | female | 33.44 | 1 | no | southeast | 8240.5896 |
| 7 | 37 | female | 27.74 | 3 | no | northwest | 7281.5056 |
| 8 | 37 | male | 29.83 | 2 | no | northeast | 6406.4107 |
| 9 | 60 | female | 25.84 | 0 | no | northwest | 28923.1369 |
| 10 | 25 | male | 26.22 | 0 | no | northeast | 2721.3208 |

**Insurance Charges Dataset**

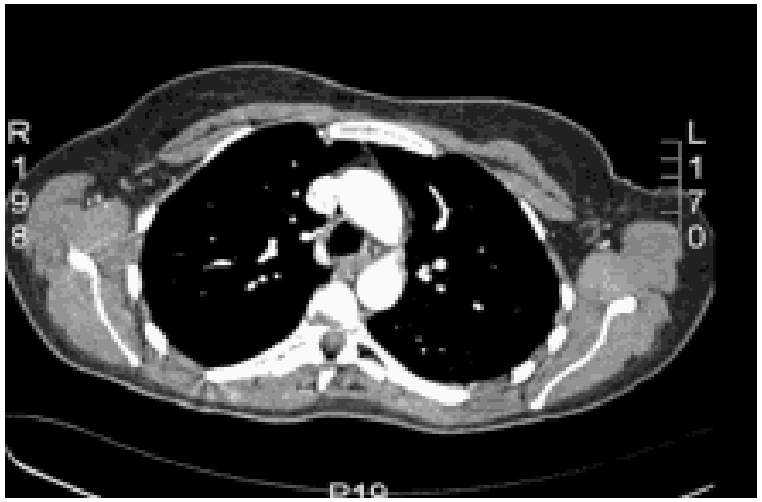
# Deep Learning Data

- Nature of Deep Learning Data:
  - Deep learning models operate on **raw or minimally processed** data
  - Unlike Machine Learning, features are often **not handcrafted/engineered**.
  - Models **learn representations** directly from dats

**Ultrasound image (2D Matrix)**




**ECG Signal. (1D vector)**



**Computed Tomography (3D Matrix)**

# Why NumPy?

- **Why Python Lists Are Not Enough?**
  - Machine learning requires operations on **large vectors, matrices, and tensors**
  - Typical operations include:
    - Element-wise arithmetic
    - Vector and matrix multiplication
    - Reductions such as sum, mean, and norms


  - These operations must be **expressive and efficient.**
  - List often require **explicit loops** for these operations (verbose, error-prone)
  - Lists stores **pointers** to objects, incurring significant **compute** and **memory overhead**

**NumPy provides numerical computational functionality with operations in mathematical form, offering performance, clarity!**

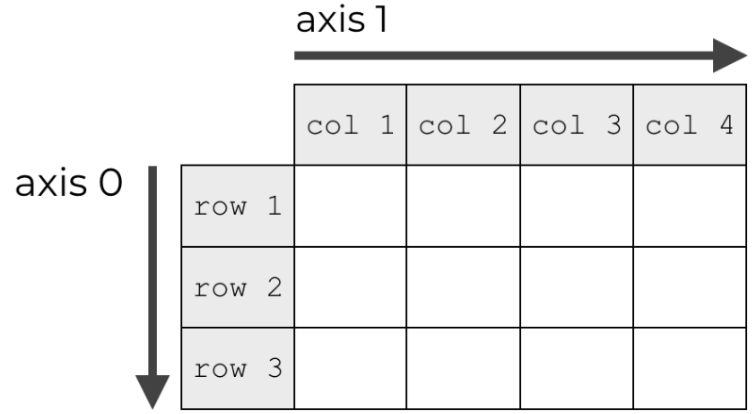
# NumPy Overview Roadmap

1. NumPy arrays for numerical computing
2. Creation and initialization of an array
3. Indexing and slicing
4. Indexing and slicing with arrays of indices
5. Iterating in arrays
6. Arithmetic operations
7. Multiplication, matrix multiplication, and dot product
8. Aggregate measures
9. Relational operators
10. Logical (bitwise) operators
11. Indices of elements that satisfy a condition & Sorting

# 1. NumPy arrays for numerical computing

- The **ndarray** implements a multi-dimensional array object and serves as the core data container for all NumPy operations.
- **Universal functions (ufuncs)** operate automatically on ndarray in an element-by-element manner, without requiring explicit loops.

- Structure and Properties of a NumPy array (**ndarray**)
  - A **ndarray** represents data arranged along one or more dimensions (**ndim**)
  - Each dimension has a length, forming the array shape (**shape**)
  - The total number of elements in the array is given by (**size**)
  - All elements in a NumPy array share the same data type (**dtype**)
  - Array values are stored in a contiguous memory buffer (**data**)
  - NumPy supports multiple numerical data types
    - Common integer types include: **int8, int16, int32, int64**
    - Common floating-point types include: **float32, float64**




**Ndim and Axes**

# 1. NumPy arrays for numerical computing

```python
# Importing NumPy as np
import numpy as np

# Define a 2D array using a Python list
array = np.array([[1, 2, 3], [4, 5, 6]])

# Access the array's attributes
ndim = array.ndim            # Number of dimensions
shape = array.shape          # Shape of the array
size = array.size            # Total number of elements
dtype = array.dtype          # Data type of elements
data = array.data            # Memory buffer data

# Print the results
print("Array:\n", array)
print("\nNumber of dimensions (ndim):", ndim)
print("Shape of the array:", shape)
print("Total number of elements (size):", size)
print("Data type (dtype):", dtype)
print("Memory buffer data:", data)
```

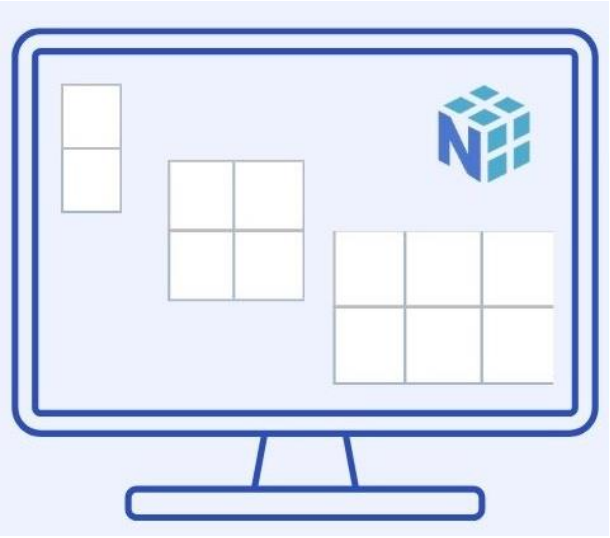
**Sample Code**

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

Number of dimensions (ndim): 2
Shape of the array: (2, 3)
Total number of elements (size): 6
Data type (dtype): int64
Memory buffer data: <memory at 0x7fd4dcb5d8c0>
```

**Output**

# 2. Creation and initialization of ndarray

- **From Python List or Tuple**: Create an array using **np.array()** with a list or tuple.
- **Zero-filled Array**: Use **np.zeros()** to create an array filled with zeros, with a defined shape.
- **One-filled Array**: Use **np.ones()** to create an array filled with ones, with a defined shape.
- **Constant-filled Array**: Use **np.full()** to create an array filled with a constant value.
- **Identity Matrix**: Use **np.eye()** to create an identity matrix of a specified size.
- **Arbitrarily-filled Array**: Use **np.empty()** to create an uninitialized array with arbitrary values.
- **Range of Values**: Use **np.arange()** to create an array with a specified range of values.
- **Linearly spaced values**: Use **np.linspace()** to create an array of linearly spaced values between a start and stop point.
- **Array Reshaping**: Use **np.reshape()** to change the shape of an array without changing its data.
- **Random Array Creation**: Use **np.random** to create arrays filled with random numbers from various distributions.



# 2. Creation and initialization of ndarray

```python
import numpy as np

# 1. Create an array from a Python list
list_array = np.array([1, 2, 3, 4])
print("Array from list:", list_array)

# 2. Create a zero-filled array with a defined shape
zero_array = np.zeros((2, 3))  # 2 rows, 3 columns
print("\nZero-filled array:", zero_array)

# 3. Create a one-filled array with a defined shape
one_array = np.ones((2, 2))  # 2x2 array
print("\nOne-filled array:", one_array)

# 4. Create a constant-filled array with a defined shape
constant_array = np.full((3, 2), 7)  # 3 rows, 2 columns, filled with 7
print("\nConstant-filled array:", constant_array)

# 5. Create an identity matrix of a defined size
identity_matrix = np.eye(3)  # 3x3 identity matrix
print("\nIdentity matrix:", identity_matrix)
```

**Sample Code**

```
Array from list: [1 2 3 4]

Zero-filled array: [[0. 0. 0.]
 [0. 0. 0.]]

One-filled array: [[1. 1.]
 [1. 1.]]

Constant-filled array: [[7 7]
 [7 7]
 [7 7]]

Identity matrix: [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

Arbitrarily-filled array: [[ 1.  2.  3.]
 [ 4.  5.  6.]]  # (values will vary as it's u .tialized)
```

**Output**

# 2. Creation and initialization of ndarray obj

```python
# 6. Create an arbitrarily-filled array with np.empty (uninitialized values)
empty_array = np.empty((2, 3))  # 2x3 array (uninitialized values)
print("\nArbitrarily-filled array:", empty_array)

# 7. Create an array with a range of values using np.arange
range_array = np.arange(1, 6)  # Values from 1 to 5
print("\nRange array:", range_array)

# 8. Create an array of linearly spaced values using np.linspace
linspace_array = np.linspace(0, 10, 5)  # 5 equally spaced values from 0 to 10
print("\nLinearly spaced array:", linspace_array)

# 9. Reshape an existing array using np.reshape
reshaped_array = np.arange(1, 7).reshape((2, 3))  # Reshape to 2x3
print("\nReshaped array:", reshaped_array)

# 10. Create a random array using np.random
random_array = np.random.random((2, 2))  # Random values between 0 and 1
print("\nRandom array:", random_array)
```

**Sample Code**

```
Arbitrarily-filled array: [[ 1.  2.  3.]
 [ 4.  5.  6.]]  # (values will vary as it's uninitialized)

Range array: [1 2 3 4 5]

Linearly spaced array: [ 0.   2.5  5.   7.5 10. ]

Reshaped array: [[1 2 3]
 [4 5 6]]

Random array: [[0.25321745 0.82786819]
 [0.05314744 0.56628569]]  # (values will vary as they are random)
```

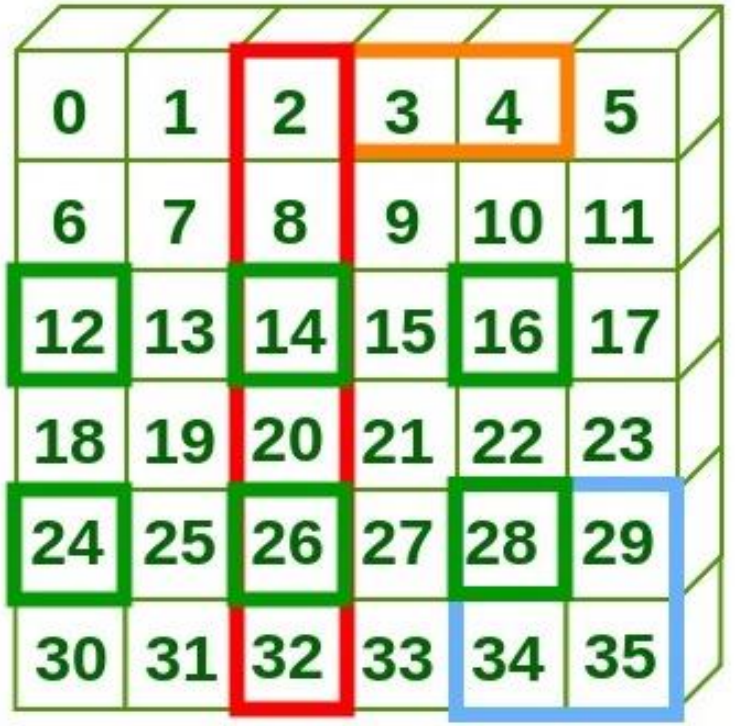
**Output**

# 3. Indexing and slicing

- **Indexing**: Accessing a specific element in a NumPy array, using row/column indices
  - Indexing syntax differs from lists (i.e., array[i, j] for **ndarray** vs array[i][j] for **lists**).

- **Slicing**: Extracting a subset of an array, by specifying start, stop, and step values for each axis.

- **Multi-dimensional ndarrays**: Both indexing and slicing can be applied to arrays with multiple dimensions (e.g., 2D, 3D).



# 3. Indexing and slicing

```python
import numpy as np

# Create a 2D array (3x3 array)
array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Indexing: Access a specific element at row 1, column 2 (value 6)
element = array[1, 2]
print("Accessed Element:", element)

# Indexing: Access an entire row (row 0)
row = array[0]
print("\nAccessed Row:", row)

# Indexing: Access an entire column (column 1)
column = array[:, 1]
print("\nAccessed Column:", column)

# Slicing: Get a subset of the array (rows 0-1, columns 0-2)
subset = array[0:2, 0:2]
print("\nSliced Subset (0:2 rows, 0:2 columns):\n", subset)

# Slicing: Get every second element from the array
sliced_every_other = array[::2, ::2]  # Take every 2nd row and column
print("\nSliced Every Other Element:\n", slice ↓ very_other)
```

**Sample Code**

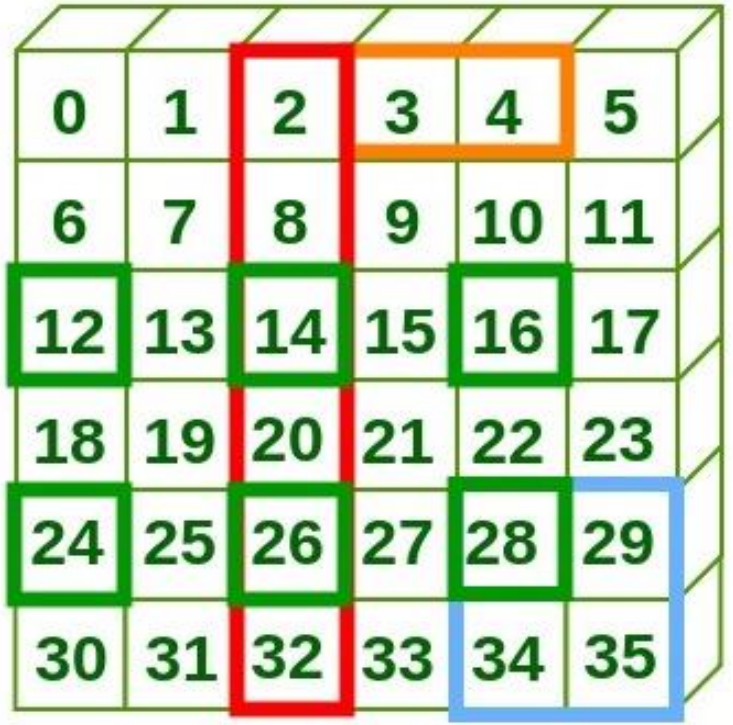
```
Accessed Element: 6

Accessed Row: [1 2 3]

Accessed Column: [2 5 8]

Sliced Subset (0:2 rows, 0:2 columns):
 [[1 2]
 [4 5]]

Sliced Every Other Element:
 [[1 3]
 [7 9]]
```

**Output**

# 4. Indexing and slicing with arrays of indices

- **Index Arrays**: Instead of using a single index, we can use an array of indices to access multiple elements or rows/columns simultaneously in multi-dimensional arrays.

- **Boolean Masks**: Boolean indexing allows you to filter or select elements in an array based on a condition, where **True** values correspond to selected elements.



# 4. Indexing and slicing with arrays of indices

```python
import numpy as np

# Create a 2D array (4x4 array)
array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16]])

# -------------------
# 1. Index Arrays (Directly defined as a 2D array)
# Define the index array directly: first row 0,2 and second row 1,3
index_array = np.array([[0, 2], [1, 3]])

# Use the index array to access elements directly
indexed_elements = array[index_array]
print("Indexed Elements (using 2D index array):", indexed_elements)

# -------------------
# 2. Boolean Masking
# Directly define a Boolean mask array (True for selected elements)
boolean_mask = np.array([[True, False, False, False],
                         [False, True, False, False],
                         [False, False, True, False],
                         [False, False, False, True]])

# Use the Boolean mask to index the original array
masked_elements = array[boolean_mask]
print("\nMasked Elements (using Boolean mask):", masked_elements)
```

Sample Code

```
Indexed Elements (using 2D index array): [ 3  8]

Masked Elements (using Boolean mask): [ 1  6 11 16]
```

Output

# 5. Iterating in arrays

- **.flat**:
  - Returns a **flat iterator** over the array.
  - It provides a **one-dimensional view** of the array, but it's **not a copy**
  - Useful for accessing or iterating over all elements in the array without altering the original structure.
- **.ravel()**:
  - Returns a **flattened array** (1D) from the original array, but unlike .flat(), **ravel()** creates a **view** whenever possible (i.e., no memory copy).
  - It is more efficient for operations where you want a flattened view of the array and will **modify the result**.

**A view in NumPy is a lightweight reference to the original array,**
**sharing its data, where modifications affect both the view and the original array.**

# 5. Iterating in arrays

```python
import numpy as np

# Create a 2D array (4x4 array)
array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16]])

# ------------------
# 1. Flatten using .flat (Iterator)
flat_array = array.flat  # .flat returns an iterator, cannot assign values
print("Flattened Array using .flat:")
for value in flat_array:
    print(value, end=" ")

print("\n")

# ------------------
# 2. Flatten using .ravel() (View or Copy)
ravel_array = array.ravel()  # Creates a view (or copy if needed)
print("\nFlattened Array using .ravel():", ravel_array)

# ------------------
# 3. Flatten using .flatten() (Always Creates a Copy)
flatten_array = array.flatten()  # Creates a copy
print("\nFlattened Array using .flatten():", flatten_array)
```

**Sample Code**

```
Flattened Array using .flat:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Flattened Array using .ravel(): [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]

Flattened Array using .flatten(): [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
```

**Output**

# 6. Arithmetic operations

- Perform operations on each element of the array individually. These operations are applied **element by element** and return a **new array**.
  - **Addition**: +
  - **Subtraction**: -
  - **Multiplication**: *
  - **Division**: /
  - **Exponentiation**: **

- **In-place Operations**: Modify the array in-place, meaning the original array is updated directly without creating a new one.
- **In-place Addition**: +=
- **In-place Subtraction**: -=
- **In-place Multiplication**: *=
- **In-place Division**: /=

# 6. Arithmetic operations (not in-place)

```python
import numpy as np

# Create a smaller 2D array (3x3 array)
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# -------------------
# 1. Array + Element (Scalar addition)
array_plus_element = array + 2
print("Array + 2 (array + element):\n", array_plus_element)

# 2. Array - Array (Array subtraction)
array2 = np.array([[1, 1, 1],
                   [1, 1, 1],
                   [1, 1, 1]])
array_minus_array = array - array2
print("\nArray - Array (array - array):\n", array_minus_array)

# 3. Array * Element (Array times scalar)
array_times_element = array * 3
print("\nArray * 3 (array * element):\n", array_times_element)

# 4. Array / Array (Array divided by another array)
array_divided_by_array = array / array2
print("\nArray / Array (array / array):\n", array_divided_by_array)
```

**Sample Code**

```
Array + 2 (array + element):
 [[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

Array - Array (array - array):
 [[0 1 2]
 [3 4 5]
 [6 7 8]]

Array * 3 (array * element):
 [[ 3  6  9]
 [12 15 18]
 [21 24 27]]

Array / Array (array / array):
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

**Output**

# 6. Arithmetic operations (in-place)

```
# --------------------
# In-place Operations

# In-place addition
array += 2
print("\nIn-place addition (array += 2):\n", array)

# Reset array to original for next operations
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# In-place subtraction
array -= array2
print("\nIn-place subtraction (array -= array2):\n", array)

# Reset array to original for next operations
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# In-place multiplication
array *= 3
print("\nIn-place multiplication (array *= 3):\n", array)
```

**Sample Code**

```
In-place addition (array += 2):
 [[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

In-place subtraction (array -= array2):
 [[2 3 4]
 [5 6 7]
 [8 9 10]]

In-place multiplication (array *= 3):
 [[ 6  9 12]
 [15 18 21]
 [24 27 30]]
```

**Output**

# 7. Multiplication, matrix multiplication, & dot product

- **Element-wise Multiplication**: Each element in one array is multiplied by the corresponding element in another array using the * operator.

- **Dot Product**: Computes the sum of the product of two vectors. It's used for vector dot products and can be done with np.dot().

- **Matrix Multiplication**: Follows linear algebra rules for matrix multiplication, where the number of columns in the first matrix must equal the number of rows in the second. This can be done using np.matmul() or the @ operator.

# 7. Multiplication, matrix multiplication, & dot product

```
# Define the arrays
A = np.array([[1, 1], [0, 1]])
B = np.array([[2, 0], [3, 4]])

# ------------------
# 1. Dot Product
# Compute the dot product of the first row of A and the second row of B
dot_product = np.dot(A[0], B[1])
print("Dot product, np.a01b1:\n", dot_product)

# ------------------
# 2. Element-wise Product
# Compute the element-wise product of A and B
elementwise_product = A * B
print("\nElement-wise product, np.a * np.b:\n", elementwise_product)

# ------------------
# 3. Matrix Product Operator (@)
# Compute the matrix product of A and B using the @ operator
matrix_product_operator = A @ B
print("\nMatrix product using @ operator, np.a @ np.b:\n", matrix_product_operator)

# ------------------
# 4. Matrix Product with np.matmul()
# Compute the matrix product of A and B using np.matmul
matrix_product_method = np.matmul(A, B)
print("\nMatrix product with np.matmul(), np.matmul(np.a, np.b):\n", matrix_product_method)
```

**Sample Code**

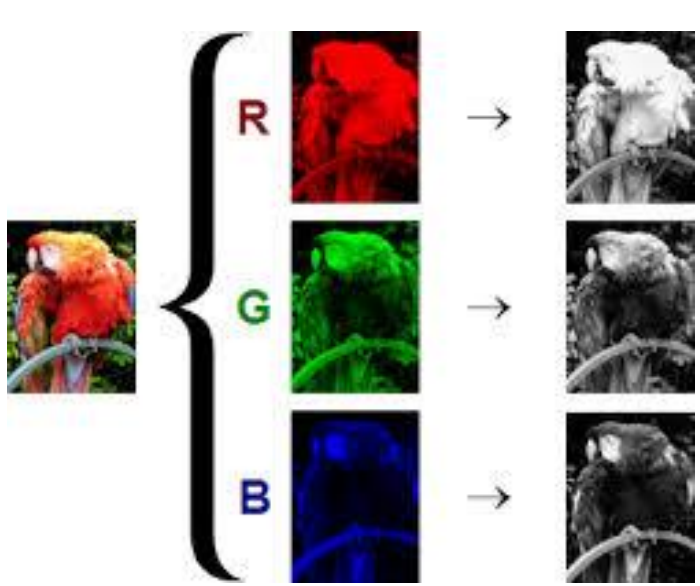
```
Dot product, np.a01b1:
 4

Element-wise product, np.a * np.b:
 [[2 0]
 [0 4]]

Matrix product using @ operator, np.a @ np.b:
 [[ 5  4]
 [ 3  4]]

Matrix product with np.matmul(), np.matmul(np.a, np.b):
 [[ 5  4]
 [ 3  4]]
```

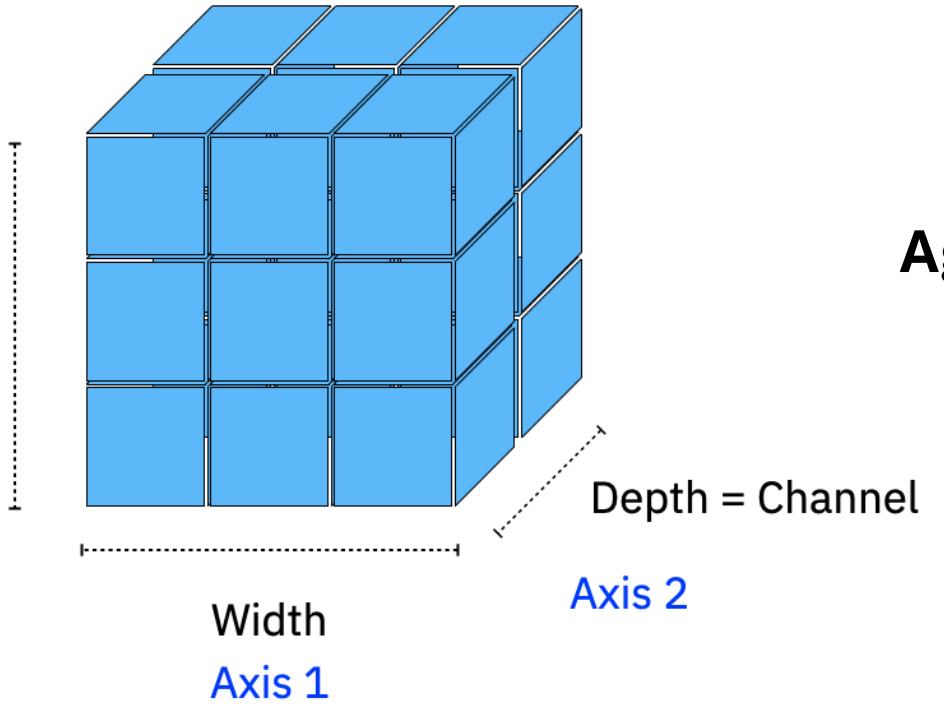
**Output**

# 8. Aggregate measures

- Operate element-wise across NumPy arrays (and, when applicable, along specified axes), enabling efficient aggregation over large datasets.
- Aggregator Functions:
  - **sum()**:
    Computes the sum of all elements in an array (or along a specified axis).
  - **min()**:
    Computes the minimum value of an array (or along a specified axis).
  - **max()**:
    Computes the maximum value of an array (or along a specified axis).
  - **argmin()**:
    Returns the index of the minimum value in the array (or along a specified axis).
  - **argmax()**:
    Returns the index of the maximum value in the array (or along a specified axis).
  - **average()**:
    Computes the weighted average of an array.
  - **median()**:
    Computes the median of an array, a robust statistic against outliers.

# 8. Aggregate measures




**Shape = (H,W,C)**

**Shape = (3,3,2)**

**Aggregate Across Axes:**

**Axes 0: (3,2)**

**Axes 1: (3,2)**

**Axes 2: (3,3)**

# 8. Aggregate measures

```python
import numpy as np

# Create a 2D array (3x3 array)
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# ------------------
# 1. Sum (with axis)
# Sum of all elements in the array
sum_all = np.sum(array)
print("Sum of all elements:", sum_all)

# Sum along axis 0 (columns)
sum_axis_0 = np.sum(array, axis=0)
print("\nSum along axis 0 (columns):\n", sum_axis_0)

# Sum along axis 1 (rows)
sum_axis_1 = np.sum(array, axis=1)
print("\nSum along axis 1 (rows):\n", sum_axis_1)

# ------------------
# 2. Min (normal application)
min_all = np.min(array)
print("\nMinimum of all elements:", min_all)
```

```python
# ------------------
# 3. Max (normal application)
max_all = np.max(array)
print("\nMaximum of all elements:", max_all)

# ------------------
# 4. Argmin (normal application)
argmin_all = np.argmin(array)
print("\nIndex of the minimum value:", argmin_all)

# ------------------
# 5. Argmax (normal application)
argmax_all = np.argmax(array)
print("\nIndex of the maximum value:", argmax_all)

# ------------------
# 6. Average (normal application)
average_all = np.average(array)
print("\nAverage of all elements:", average_all)

# ------------------
# 7. Median (normal application)
median_all = np.median(array)
print("\nMedian of all elements:", median_all)
```

**Sample Code**

```
Sum of all elements: 45

Sum along axis 0 (columns):
 [12 15 18]

Sum along axis 1 (rows):
 [ 6 15 24]

Minimum of all elements: 1

Maximum of all elements: 9

Index of the minimum value: 0

Index of the maximum value: 8

Average of all elements: 5.0

Median of all elements: 5.0
```

**Output**

# 9. Relational Operators

- **Element-wise Comparison**: Compare an **array** to a **single element** (scalar), producing a **Boolean array** as the result.

- **Array-to-Array Comparison**: Operators can also be applied **between two arrays**, performing **element-wise comparisons** and returning a Boolean array indicating the result of each comparison.

- **Common Relational Operators**:
  - **Greater than or equal to (>=)**
  - **Less than or equal to (<=)**
  - **Equal to (==)**
  - **Not equal to (!=)**

# 9. Relational Operators

```python
import numpy as np

# Create a 2D array (3x3 array)
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# -------------------
# 1. Greater than or equal to (array compared with an element)
greater_than_or_equal = array >= 5
print("Array >= 5 (array compared with an element):\n", greater_than_or_equal)

# -------------------
# 2. Less than or equal to (array compared with another array)
array2 = np.array([[2, 3, 4],
                   [5, 6, 7],
                   [8, 9, 10]])
less_than_or_equal = array <= array2
print("\nArray <= Array2 (array compared with another array):\n", less_than_or_equal)

# -------------------
# 3. Equal to (array compared with an element)
equal_to_element = array == 6
print("\nArray == 6 (array compared with an element):\n", equal_to_element)

# -------------------
# 4. Not equal to (array compared with another array)
not_equal_to_array = array != array2
print("\nArray != Array2 (array compared with another array):\n", not_equal_to_array)
```

**Sample Code**

```
Array >= 5 (array compared with an element):
 [[False False False]
 [False  True  True]
 [ True  True  True]]

Array <= Array2 (array compared with another array):
 [[ True  True  True]
 [ True  True  True]
 [ True  True  True]]

Array == 6 (array compared with an element):
 [[False False False]
 [False False  True]
 [False False False]]

Array != Array2 (array compared with another array):
 [[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

**Output**

# 10. Logical (bitwise) Operators

- **Bitwise logical operators** work on **boolean arrays** that result from relational comparisons and allow **efficient operations** on these arrays.

- Common Operators:
  - **& (and)**: Performs **AND** operation on **boolean**
  - **| (or)**: Performs **OR** operation on **boolean arrays**
  - **~ (not)**: Performs **NOT** operation, flipping the bits of a boolean array.
  - **^ (xor)**: Performs **XOR** operation, which returns True only when the inputs differ.

# 10. Logical (bitwise) Operators

```python
import numpy as np

# Create a 2D array (3x3 array)
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# -------------------
# 1. Relational Operation: Compare with an element (e.g., 5)
relational_result = array >= 5  # Element-wise comparison with 5
print("Array >= 5 (relational comparison with element):\n", relational_result)

# -------------------
# 2. Logical (Bitwise) Operation: Apply logical AND (&) and OR (|)
logical_and = relational_result & (array % 2 == 0)  # AND with a condition (even numbers)
print("\nLogical AND with condition (array >= 5 and even numbers):\n", logical_and)

logical_or = relational_result | (array % 2 != 0)  # OR with a condition (odd numbers)
print("\nLogical OR with condition (array >= 5 or odd numbers):\n", logical_or)

# -------------------
# 3. Logical NOT (~): Flip the boolean values
logical_not = ~relational_result  # NOT operation (invert the boolean values)
print("\nLogical NOT (invert relational result):\n", logical_not)
```

**Sample Code**

```
Array >= 5 (relational comparison with element):
 [[False False False]
 [ True  True  True]
 [ True  True  True]]

Logical AND with condition (array >= 5 and even numbers):
 [[False False False]
 [ True  True  True]
 [False  True  True]]

Logical OR with condition (array >= 5 or odd numbers):
 [[False  True  True]
 [ True  True  True]
 [ True  True  True]]

Logical NOT (invert relational result):
 [[ True  True  True]
 [False False False]
 [False False False]]
```

**Output**

# 11. Indices of elements that satisfy a condition & Sorting

- **Finding Indices with np.where**: Returns the indices of the elements in an array that satisfy a given condition. It can be used in two forms:
  - **np.where(condition)**: Returns the indices where the condition is **True**.
  - **np.where(condition, x, y)**: Selects values from x where the condition is **True**, and from y otherwise.

- **Sorting an Array**:
  - **In-place Sorting**: Use **a.sort()** to sort the array in **place**, meaning the original array is modified and no new array is returned.
  - **Not in-place Sorting**: Use **np.sort(a)** to return a **new sorted array**, leaving the original array unchanged

# 11. Indices of elements that satisfy a condition & Sorting

```python
import numpy as np

# Create a 2D array (3x3 array)
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# -------------------
# 1. Finding Indices with np.where
# Find the indices where the elements are greater than 5
indices = np.where(array > 5)
print("Indices of elements > 5:\n", indices)

# -------------------
# 2. Using np.where with condition, x, y
# If the element is greater than 5, set it to 10, otherwise set it to 0
modified_array = np.where(array > 5, 10, 0)
print("\nModified array with np.where (condition, x, y):\n", modified_array)

# -------------------
# 3. Sorting the Array
# Not in-place sorting using np.sort (creates a new sorted array)
sorted_array = np.sort(array, axis=None)
print("\nSorted array (not in-place):\n", sorted_array)

# In-place sorting using a.sort()
array.sort(axis=None)
print("\nIn-place sorted array (array.sort()):\n", array)

# -------------------
# 4. Sorting with axis (example for rows)
sorted_by_rows = np.sort(array, axis=1)  # Sort each row
print("\nSorted array by rows (not in-place):\n", sorted_by_rows)

# -------------------
# 5. Sorting with axis (example for columns)
sorted_by_columns = np.sort(array, axis=0)  # Sort each column
print("\nSorted array by columns (not in-place):\n", sorted_by_columns)
```

**Sample Code**

```
Indices of elements > 5:
 (array([1, 2, 2, 2]), array([2, 0, 1, 2]))

Modified array with np.where (condition, x, y):
 [[ 0  0  0]
  [ 0  0 10]
  [10 10 10]]

Sorted array (not in-place):
 [1 2 3 4 5 6 7 8 9]

In-place sorted array (array.sort()):
 [[1 2 3]
  [4 5 6]
  [7 8 9]]

Sorted array by rows (not in-place):
 [[1 2 3]
  [4 5 6]
  [7 8 9]]

Sorted array by columns (not in-place):
 [[1 2 3]
  [4 5 6]
  [7 8 9]]
```

**Output**

**I hope you found this session useful.**
**Keep exploring NumPy!**