- Missing Topics
    - Resize, and reshape
    - Broadcasting

Numpy

- Designed for scientific computation
- NumPy is faster than regular Python lists
- In short:
    - NumPy moves the heavy lifting from slow Python code to fast, optimized C code working on compact memory buffers — that's why it's much faster for numerical computations.

Feature	Python List	Python array.array	NumPy Array
Data type	Can store mixed data types	Stores only one data type (e.g., 'i' for int)	Stores homogeneous data types (int, float, etc.)
Flexibility	Very flexible, heterogeneous	Less flexible, homogeneous only	Very flexible but homogeneous
Performance	Slower for numeric computations	Faster than list for numeric data	Much faster due to C implementation & vectorized ops
Memory	More memory overhead per element	More compact than lists	Most memory efficient, contiguous memory block
Operations	No built-in vectorized operations	Limited to simple numeric ops	Rich set of vectorized mathematical and logical operations
Supported Dimensions	1D and nested lists (for multi-dim)	1D only	Multi-dimensional (nD arrays)
Use cases	General-purpose container	Efficient numeric storage in 1D	Scientific computing, data analysis, ML, etc.

- Core object: ndarray — N-dimensional array.
- In Numpy dimensions are called axes. The number of axes is called rank
    - [[0.,0.,0.,0.],
     [0.,0.,0.,0.],
     [0.,0.,0.,0.]]
    - It has 2 dimensions, first dimension's length is 4 (4 elements)
    - Second dimension length is 3

- Create array: np.array([1,2,3])
- Attributes:
    - .shape (tuple of dimensions)
    - .size (total elements)

- .ndim (number of dimensions)
- .dtype (data type of elements)
- arr = np.array([[1, 2, 3], [4, 5, 6]])
  print(arr.shape) # (2, 3)
  print(arr.size)   # 6
  print(arr.ndim)   # 2
  print(arr.dtype) # int64 (or int32 depending on system)

- Array Creation
  - np.zeros((m, n)) — create m x n array filled with zeros
    arr_zeros = np.zeros((3, 4))
    print("Zeros:\n", arr_zeros)

  - np.ones((m, n)) — create m x n array filled with ones
    arr_ones = np.ones((2, 5))
    print("\nOnes:\n", arr_ones)

  - np.full((m, n), val) — create m x n array filled with a specified value
    arr_full = np.full((2, 3), 7)
    print("\nFull with 7:\n", arr_full)

  - np.empty((m, n)) — create m x n uninitialized array (values are arbitrary, fast to create)
    arr_empty = np.empty((2, 4))
    print("\nEmpty (uninitialized):\n", arr_empty)

  - np.arange(start, stop, step) — create array with evenly spaced values (like range)
    arr_arange = np.arange(0, 10, 2)
    print("\nArange (0 to 10 step 2):\n", arr_arange)

  - np.linspace(start, stop, num) — create array with num evenly spaced values between start and stop
    arr_linspace = np.linspace(0, 1, 5)
    print("\nLinspace (5 points between 0 and 1):\n", arr_linspace)

  - np.logspace(start, stop, num) — num points spaced evenly on a log scale (10**start to 10**stop)
    arr_logspace = np.logspace(1, 3, 4) # 10^1 to 10^3 in 4 steps
    print("\nLogspace (4 points between 10^1 and 10^3):\n", arr_logspace)

  - np.eye(n) — create n x n identity matrix (1s on diagonal, 0 elsewhere)
    eye_matrix = np.eye(4)
    print("\nIdentity matrix (eye):\n", eye_matrix)

- np.identity(n) — same as np.eye(n), create identity matrix
  identity_matrix = np.identity(3)
  print("\nIdentity matrix (identity):\n", identity_matrix)

- np.fromfunction(func, shape) — create array by applying function to each coordinate
  def my_func(i, j):
     return i + j

  arr_from_func = np.fromfunction(my_func, (3, 3), dtype=int)
  print("\nArray from function (i + j):\n", arr_from_func)

- Indexes in Numpy array starts with 0
- My_array[1,3] —>> element in second row, fourth column
- In both NumPy and Pandas, axis 0 refers to rows, which is the vertical direction, and axis 1 refers to columns, the horizontal direction
- 1D, 2D, 3D arrays
  - 1D
    - Array = [0,1,2,3,4,5]
    - Array[2] = 2
  - 2D array is array of 1D arrays
    - Two_dim_array = [[1,2,3],[4,5,6],[7,8,9]]
    - Two_dim_array[1,2] ——->>6
  - 3D array
    - [
          [
              [1,2,3,4],
              [5,6,7,8]
          ],
          [
              [9,10,11,12],
              [13,14,15,16]
          ]
      ]
    - 3d_array[0,1,2]---->> 7

- Reshaping arrays
  - a= np.arange(6)
    print(a)
    [0,1,2,3,4,5]
    b = a.reshape(2,3)
    print(b)
    [[0,1,2],[3,4,5]]

- Addition in Numpy arrays
    - They apply element wise
    - a= np.array([20,30,40,50])
      b = np.arange(4)----->> array([0,1,2,3])
      c = a+b —->>> array([20,31,42,53]
- Subtraction - element wise
- Element wise Product in Numpy
    - A = np.array([[1,1],[0,1]]
    - B = np.array([[2,0],[3,4]]
    - A*B—>> element wise product
        - [[2,0],[0,4]]
- Matrix product in Numpy arrays :
    - A = np.array([[1,1],[0,1]])
    - B = np.array([[2,0],[3,4]])
    - np.dot(A,B)
        - Output —-> [[5,4],[3,4]]
- Division in Numpy arrays :
    - They apply element wise
    - A = np.array([20,30,40,50])
    - B = np.arange(1,5)
        - Output —> [1,2,3,4]
    - C = a/b
    - Output —-> [20.,15., 13.33, 12.5]
- Integer Division
- Modulus
- Exponents in Numpy arrays :
    - It is applied element wise
    - A = np.array([20,30,40,50])
    - B = np.arange(1,5)
    - C = a**b
    - Output —---> array([20, 900. 64000, 6250000])
- Conditional Operators on Numpy arrays
    - They are also applied element wise
    - M = np.array([20,-5,30,40])
    - M<[15,16,35,36]
    - Output —-> array([False, True, True, False], dtype = bool)
    - M<25
    - Output —---> array([True, True, False, False], dtype = bool)
    - To get the elements below 25,
        - M[M<25]
        - Output —---> array[20,-5]
- Ndarray slices are actually views on the same data buffer. If you modify the slice, it is going to modify the original ndarray as well

- If you want a copy of the data, you need to use the copy method
  - B = a[2:6].copy()
  - Now if we modify B, a will not change
- Numpy array and regular python array :
  - A = np.array([1,2,5,7,8])
  - A[1:3] = -1
  - Output —-->> array(1,-1,-1,-1,7,8])
  - It will not work like this for lists
- Mathematical Operations along different axes
  - ```
    arr = np.array([[1, 2, 3],
                    [4, 5, 6]])
    np.sum(arr, axis=0)  # [5, 7, 9] → sum down each column, Collapse rows
    np.sum(arr, axis=1)  # [6, 15]  → sum across each row, Collapse columns

    np.mean(arr, axis=0)  # [2.5, 3.5, 4.5]
    np.mean(arr, axis=1)  # [2.0, 5.0]

    np.max(arr, axis=0)  # [4, 5, 6]
    np.max(arr, axis=1)  # [3, 6]
    ```

Think of axis as the direction you move:
- axis=0 → move down rows → operate on columns
- axis=1 → move across columns → operate on rows
    ```