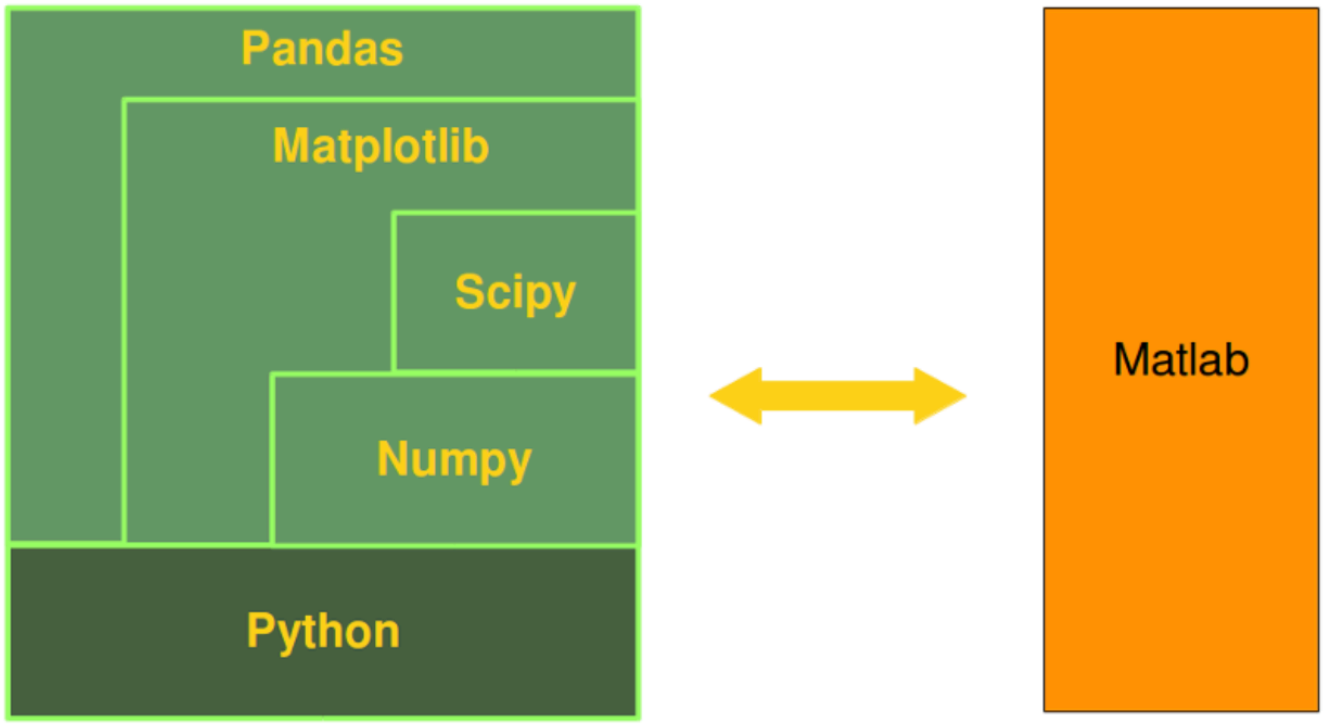
-**Numpy** (numeric python) is a module which provides the basic data structures, implementing multi-dimensional arrays and matrices.

-**SciPy** (Scientific Pyhthon) is based on top of Numpy. It
extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.

-**Matplotlib** is a plotting library for the Python programming
language and the numerically oriented modules like NumPy and
SciPy.

-**Pandas** = Panel Datas (The youngest child) used for data structure, data manipulating, and analysis. Well suited with tabular data (spreadsheet programming like Excel)

PYTHON, AN ALTERNATIVE TO MATLAB*



1, **Numpy**
Advantages of using Numpy with Python:
• array oriented computing
• efficiently implemented multi-dimensional arrays
• designed for scientific computation

```
import numpy
```

But you will hardly ever see this. Numpy is usually renamed to np:

```
import numpy as np
```

Our first simple Numpy example deals with temperatures. Given is a list with values, e.g. temperatures in Celsius:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 2
0.1]
```

We will turn our list "cvalues" into a one-dimensional numpy array:

```
C = np.array(cvalues)
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

```
print(C * 9 / 5 + 32)
```

```
[68.18 69.44 71.42 72.5  72.86 72.14 71.24 70.16 69.62 68.18]
```

The array C has not been changed by this expression:

```
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Compared to this, the solution for our Python list looks awkward:

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

```
[68.18, 69.44, 71.42, 72.5, 72.86, 72.14, 71.24000000000001, 70.1
6, 69.62, 68.18]
```

So far, we referred to C as an array. The internal type is "ndarray" or to be even more precise "C is an instance of the class numpy.ndarray":

```
type(C)
```

Output:
```
numpy.ndarray
```

In the following, we will use the terms "array" and "ndarray" in most cases synonymously.

Array Creation

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

+++**NumPy**'s array class is called **ndarray**. It is also known by the **alias array**. Note that numpy.arrayis not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an **ndarrayobject** are:

-**ndarray.ndim** the number of axes (dimensions) of the array.

-**ndarray.shape** the dimensions of the array. This is a tuple of integers indicating the size of the array in each di-mension. For a matrix with n rows and m columns, **shape** will be (n,m). The length of the shapetuple is therefore the number of axes, **ndim**.

-**ndarray.size** the total number of elements of the array. This is equal to the product of the elements of shape.

-**ndarray.dtype** an object describing the type of the elements in the array. One can create or specify dtype's us-
ing standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

-**ndarray.itemsize** the size in bytes of each element of the array. For example, an array of elements of type float64
has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

-**ndarray.data** the buffer containing the actual elements of the array. Normally, we won't need to use this attribute
because we will access the elements in an array using indexing facilities.

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

-NumPy displays it in a similar way to nested lists, but with the following layout:
• the last axis is printed from left to right,
• the second-to-last is printed from top to bottom,
• the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

+++**One-dimensional** arrays are then printed as rows, **bidimensionals** as matrices and **tridimensionals** as lists of matrices.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Basic Operation

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
```

work the same with multiply, sqrt, sin(), addition,

The basics

```
a = np.ones(3, dtype=np.int32)
```

> > > b = np.linspace(0,pi,3)
> > >
> > > b.dtype.name
> > > 'float64'
> > > c = a+b
> > > c
> > > array([1. , 2.57079633, 4.14159265])
> > > c.dtype.name
> > > 'float64'
> > > d = np.exp(c*1j)
> > > d
> > > array([0.54030231+0.84147098j, -0.84147098+0.54030231j, -0.54030231-0.84147098j])
> > > d.dtype.name
> > > 'complex128'

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of
the ndarray class.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However,
by specifying the **axis** parameter you can apply an operation along the specified axis of an

array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

NumPy (Numerical Python) is the **core library for numerical computing in Python**.
Almost every ML, data science, or AI library depends on it.

NumPy is mainly used for:

- Handling **large numerical datasets**
- Performing **fast mathematical operations**
- Working with **vectors, matrices, and tensors**

Libraries like **Pandas, Scikit-learn, TensorFlow, PyTorch** all rely on NumPy arrays internally.

1. The Core Concept: `ndarray`

At the heart of NumPy is the **ndarray** (n-dimensional array).

Why ndarray is better than Python lists

Python List	NumPy Array
Can store mixed types	Stores one data type only
Slower for math	Very fast (C-based)
Manual loops	Vectorized operations

Example

```
import numpy as np
```

```
a = np.array([1, 2, 3, 4])
print(a)
```

Output:

```
[1 2 3 4]
```

3. Array Dimensions (Axes)

- **1D array** → vector
- **2D array** → matrix
- **3D+ array** → tensor

Example

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

This array has:

- `ndim = 2`
- `shape = (2, 3)` → 2 rows, 3 columns

```
b.shape      # (2, 3)
b.ndim       # 2
b.size       # 6
```

4. Why NumPy Is Fast (Vectorization)

Instead of looping in Python, NumPy performs operations **all at once**.

Python list (slow)

```
c = []
for i in range(len(a)):
    c.append(a[i] * 2)
```

NumPy (fast & clean)

```
c = a * 2
```

This is called **vectorization**.

- Shorter code
- Faster execution
- Fewer bugs

5. Broadcasting (Very Important for ML)

Broadcasting allows NumPy to operate on arrays of **different shapes**.

Example

```
a = np.array([1, 2, 3])
a + 10
```

Output:

```
[11 12 13]
```

Or

```
x = np.array([[1, 2, 3],
              [4, 5, 6]])
y = np.array([10, 20, 30])

x + y
```

Output:

```
[[11 22 33]
 [14 25 36]]
```

Note: This is heavily used in **ML feature scaling and normalization**.

6. Creating Arrays

From lists

```
np.array([1, 2, 3])
```

Zeros, ones, empty

```
np.zeros((2, 3))
np.ones((3, 2))
np.empty((2, 2))
```

Ranges

```
np.arange(0, 10, 2)       # [0 2 4 6 8]
np.linspace(0, 1, 5)      # [0.   0.25 0.5  0.75 1. ]
```

+++ `linspace` is preferred in ML when precision matters.

7. Data Types (`dtype`)

NumPy arrays store **homogeneous data types**, which improves performance.

Example

```
a = np.array([1, 2, 3])
a.dtype
```

Output:

```
int64
```

We can also specify types:

```
np.array([1, 2, 3], dtype=np.float32)
```

This is important for:

- Memory optimization
- ML models (especially deep learning)

8. Basic Array Operations

Element-wise operations

```
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])

a + b     # [11 22 33]
```

```
a * b      # [10 40 90]
a ** 2     # [100 400 900]
```

Comparisons

```
a > 15
```

Output:

```
[False True True]
```

Perfect, this is where NumPy really starts to **feel like machine learning**, so I'll go **deep and specific** from **Section 9 onward**, with **clear explanations, step-by-step logic, and practical examples** you can reuse in exams or projects.

9. Matrix Multiplication (Core of ML)

In NumPy, **matrix multiplication is NOT done with `*`**.

- `*` → element-wise multiplication
- `@` or `np.dot()` → matrix multiplication

Why this matters in ML

Almost every ML algorithm is based on:
[
y = XW + b
]
which is **matrix multiplication**.

Example: Element-wise vs Matrix Multiplication

```
import numpy as np

A = np.array([[1, 2],
              [3, 4]])

B = np.array([[5, 6],
              [7, 8]])
```

Element-wise

```
A * B
```

Output:

```
[[ 5 12]
 [21 32]]
```

Matrix multiplication

```
A @ B
```

Result:

```
[[19 22]
 [43 50]]
```

Explanation:

- Each row of `A` is multiplied with each column of `B`
- Used in **linear regression, neural networks, PCA**

ML Example: Linear Model Prediction

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

W = np.array([[0.5],
              [1.0]])

y = X @ W
```

This represents:
[
y = 0.5x_1 + 1.0x_2
]

10. Aggregation Functions (Summarizing Data)

Aggregation functions **reduce data into meaningful statistics**.

Common ones

- `sum()`
- `mean()`
- `min()`, `max()`
- `std()`, `var()`

Example

```
data = np.array([[10, 20, 30],
                 [40, 50, 60]])
```

```
data.sum()          # 210
data.mean()         # 35
data.min()          # 10
data.max()          # 60
```

Axis-based Aggregation (VERY important)

```
data.sum(axis=0)  # column-wise
data.sum(axis=1)  # row-wise
```

Results:

```
axis=0 → [50 70 90]
axis=1 → [ 60 150 ]
```

ML meaning:

- `axis=0` → feature-wise statistics
- `axis=1` → sample-wise statistics

11. Indexing and Slicing (Accessing Data Precisely)

1D Example

```
a = np.arange(10)
a[2:7]
```

Result:

```
[2 3 4 5 6]
```

2D Indexing

```
b = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

```
b[0, 1]    # 2
b[:, 1]    # [2 5 8]
b[1, :]    # [4 5 6]
```

ML use:

- Selecting features
- Extracting labels
- Splitting datasets

12. Boolean Indexing (Filtering Data)

Boolean indexing lets you **filter data without loops**.

Example

```
scores = np.array([45, 78, 88, 62, 91])
scores[scores >= 80]
```

Output:

```
[88 91]
```

ML Example: Remove Outliers

```
data = np.array([10, 12, 11, 300, 13])
clean = data[data < 100]
```

Result:

```
[10 12 11 13]
```

This is commonly used in:

- Data cleaning
- Anomaly detection
- Feature selection

13. Shape Manipulation (Critical for ML Pipelines)

ML models expect data in **specific shapes**.

Reshape

```
a = np.arange(12)
a.reshape(3, 4)
```

You can use `-1` to auto-calculate:

```
a.reshape(2, -1)
```

Flatten

```
a.ravel()
```

Used when converting images or tensors into vectors.

Transpose

```
X.T
```

Used in:

- Covariance matrices
- Linear algebra formulas
- Backpropagation

14. Copies vs Views (Memory & Bugs)

This is a **common source of hidden bugs**.

View (shared memory)

```
a = np.array([1, 2, 3, 4])
b = a[:2]
b[0] = 99
```

Now:

```
a → [99 2 3 4]
```

Deep Copy (safe)

```
b = a[:2].copy()
```

Now modifying `b` **does not affect** `a`.

ML best practice:

Always `.copy()` when creating training/validation splits.

15. Fancy Indexing (Advanced Data Selection)

Fancy indexing allows indexing with arrays.

Example

```
a = np.array([10, 20, 30, 40, 50])
idx = [0, 2, 4]

a[idx]
```

Result:

```
[10 30 50]
```

2D Example

```
A = np.arange(12).reshape(3, 4)
A[[0, 2], [1, 3]]
```

Result:

```
[1 11]
```

Used in:

- Sampling mini-batches
- Selecting specific features

16. Linear Algebra Operations (ML Backbone)

NumPy's `linalg` module supports ML math.

Inverse

```
np.linalg.inv(A)
```

Solve system

```
np.linalg.solve(A, b)
```

Eigenvalues

```
np.linalg.eig(A)
```

Used in:

- PCA
- Optimization
- Dimensionality reduction

17. Practical ML Example (End-to-End)

Normalize Features

```
X = np.array([[50, 200],
              [60, 220],
              [55, 210]])

X_norm = (X - X.mean(axis=0)) / X.std(axis=0)
```

Prediction

```python
W = np.array([[0.3],
              [0.7]])

y_pred = X_norm @ W
```