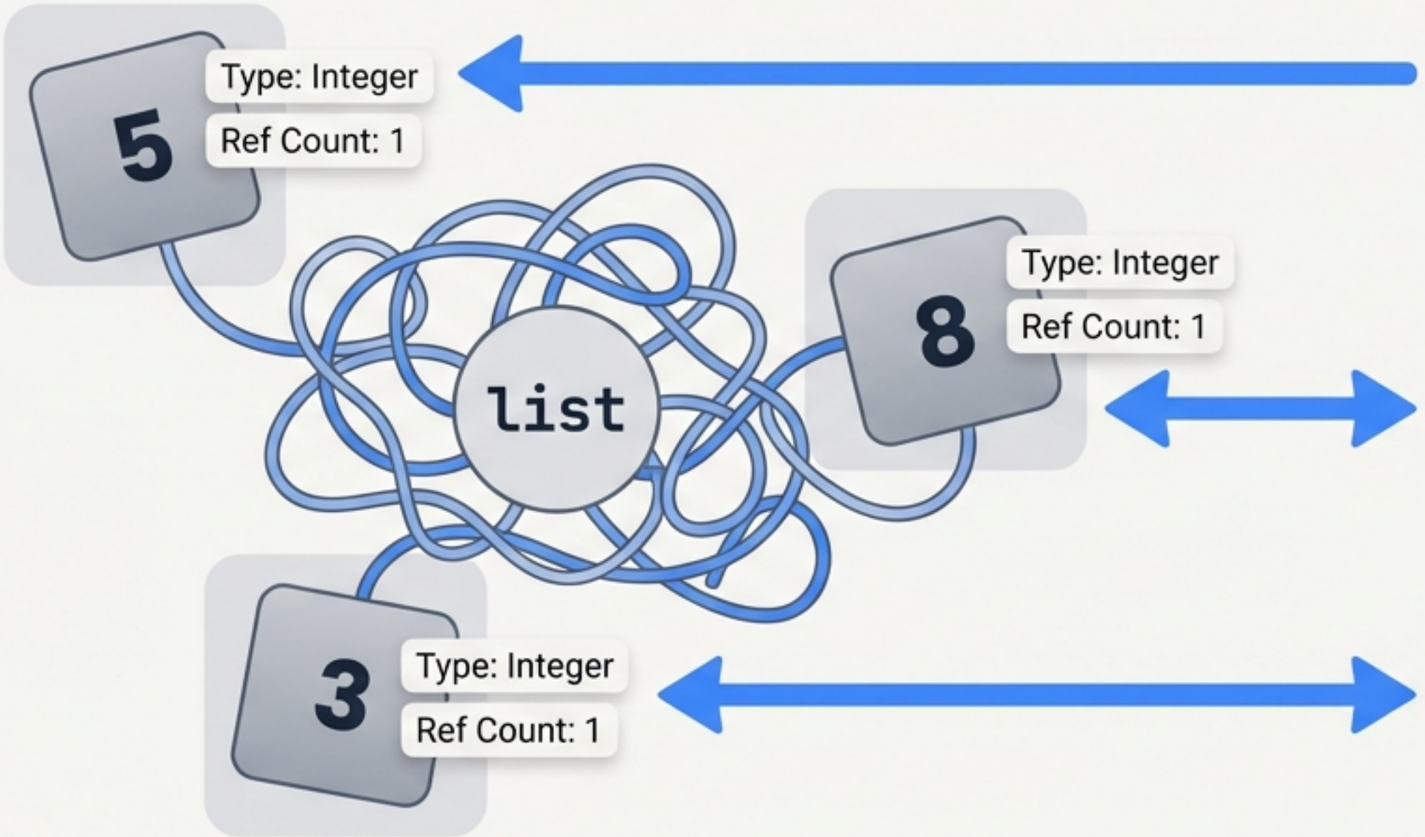
# Python’s Superpower for Numbers

## A Beginner’s Guide to Mastering NumPy

# First, Why Do We Even Need NumPy?

Standard Python is fantastic, but it wasn't built for high-speed, large-scale data crunching. Its core data types, like lists, have a fundamental problem.

## Python List: The Jumbled Toy Box




## The Problem

### Memory Overhead

Each number isn't just a number; it's a full-blown Python object with extra metadata, wasting space.

### Pointer Chase

The CPU has to follow each tangled pointer to find the next number in memory. This is slow and inefficient.

### Type Checking

For every single operation (like adding two numbers), Python has to check the type of each box.

# 🐌 The Real Cost: Python Loops are Incredibly Inefficient

```
# Adding two lists in Python
c = []
for i in range(len(a)):
    c.append(a[i] + b[i])
```

What Python does for *every single number*:

- Follow a pointer to a box.
- “Is this an integer?”
- Extract the raw number.
- Follow another pointer.
- “Is this also an integer?”
- Extract the raw number.
- Perform the actual addition.
- Create a brand new Python object (a new box) for the result.
- Add a pointer to this new box to list `c`.

This happens for millions of numbers. It’s like gift-wrapping and unwrapping every single number for every single calculation.

# NumPy to the Rescue!

NumPy solves Python's speed problem by going back to first principles with a new data type: the `ndarray`.

## Solution A: Contiguous Memory


- **A solid block:** A NumPy array is a continuous block of memory.
- **No 'boxes':** It stores the raw values directly, with no pointers or metadata overhead.
- **Homogeneous Data:** All elements *must* be the same type (e.g., all `int64`). This eliminates the need for type-checking.

### The Godfather of NumPy

Created by Travis Oliphant in 2005 to unify two competing libraries (Numeric and Numarray) and give Python a single, powerful tool for scientific computing.

# The Secret Weapon: Vectorization (SIMD)

Because the data is in one continuous block and the type is known, NumPy can use special, low-level CPU instructions called **SIMD (Single Instruction, Multiple Data)**. Instead of adding numbers one by one in a loop, the CPU can load a chunk (a “vector”) of 4 or 8 numbers at once and perform the operation in a single clock cycle.

## Python Loop

## NumPy Vectorization


Left Code Block (Python)

```python
# Inefficient Loop
for i in range(len(a)):
    c.append(a[i] + b[i])
```

Right Code Block (NumPy)

```python
# Vectorized Operation
c = a + b
```

This one line triggers the super-fast vectorized operation!

# 50–100x FASTER

VS

**Pure Python Loop**

**NumPy Vectorized Operation**

This is why NumPy is the foundation for almost every data science, machine learning, and scientific package in Python, including Pandas, Matplotlib, and Scikit-learn.

# Level 1: Creating Your First Arrays

## Step 1: The Import

```
import numpy as np
```

First, we summon our hero and give it a short name, `np`.

## Step 2: Creating Arrays from Lists

### Example 1: 1D Array (Vector)

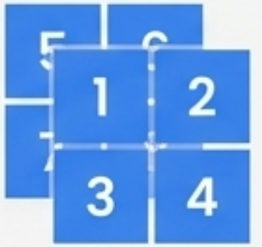
```
a = np.array([1, 2, 3])
```


### Example 2: 2D Array (Matrix)

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


### Example 3: 3D Array (Tensor)

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




Notice the square brackets! A 3D array is just a list of 2D arrays.

# The Array Creation Toolkit

Create arrays from scratch without needing a Python list first.

## np.arange(start, stop)

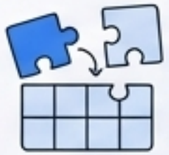
Create a sequence of numbers, just like Python's `range()`.

```
np.arange(1, 11)
```

```
[1 2 3 4 5 6 7 8 9 10]
```

## .reshape(rows, cols)

The perfect partner for `arange`! Molds your 1D array into a new shape.



```
np.arange(1, 13).reshape(3, 4)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

## np.ones((rows, cols))

Quickly make an array full of 1s. (Useful for initialization).

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

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

## np.zeros((rows, cols))

You guessed it... an array full of 0s.

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

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

## np.linspace(start, end, num)

Get `num` evenly spaced points between a `start` and `end`.

```
np.linspace(0, 10, 5)
```

```
[ 0.  2.5  5.  7.5 10. ]
```

## np.identity(n)

Create an `n × n` identity matrix in one step.

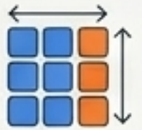
```
np.identity(3)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

# Level 2: Inspecting Your Array

## What's inside? Let's find out.

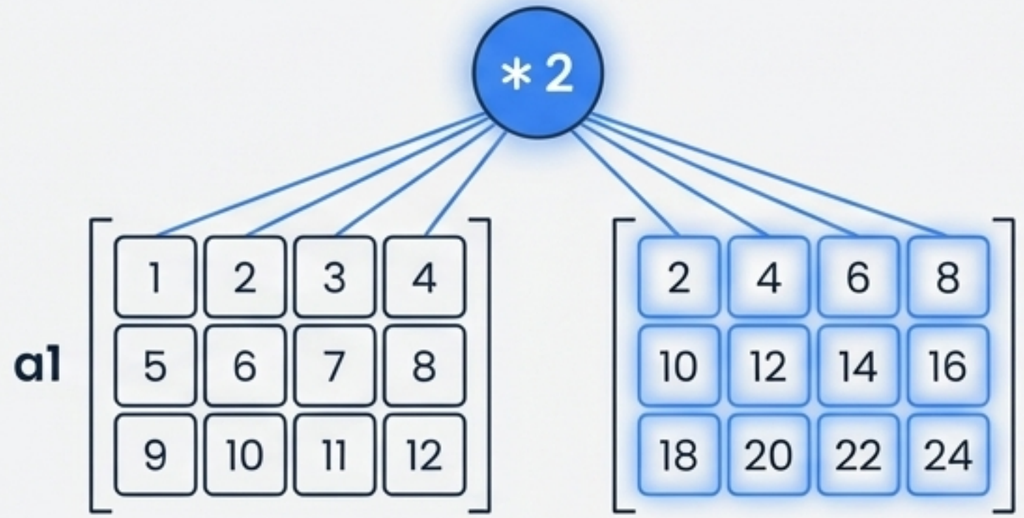
```
a = np.arange(12).reshape(3, 4)
# Our array for this section:
# [[ 0 1  2  3]
#  [ 4 5  6  7]
#  [ 8 9 10 11]]
```

| Attribute | Output | What it means |
|---|---|---|
| a.ndim | 2 | Number of dimensions. It's a 2D matrix.  |
| a.shape | (3, 4) | The size of each dimension. 3 rows, 4 columns.  |
| a.size | 12 | Total number of elements in the array (3 * 4).  |
| a.dtype | int64 | The data type of the elements. Here, 64-bit integers.   |
| a.itemsize | 8 | How much memory (in bytes) each element takes up.   |

# Operations on Entire Arrays, No Loops Needed.

## Part 1: Scalar Operations (Array + Number)

The operation is "broadcast" to every element in the array.




### a1 * 2 (Multiplication)

```
a1 * 2
```

```
array([[ 2,  4,  6,  8],
       [10, 12, 14, 16],
       [18, 20, 22, 24]])
```

### a1 + 100 (Addition)

```
a1 + 100
```

```
array([[101, 102, 103, 104],
       [105, 106, 107, 108],
       [109, 110, 111, 112]])
```

### a1 > 5 (Relational)

```
a1 > 5
```

```
array([[False,False, False,False],
       [False,True, True, True],
       [True, True, True, True]])
```

## Part 2: Vector Operations (Array + Array)

The operation is performed element-wise between the two arrays.


```
c = a1 + a2
```

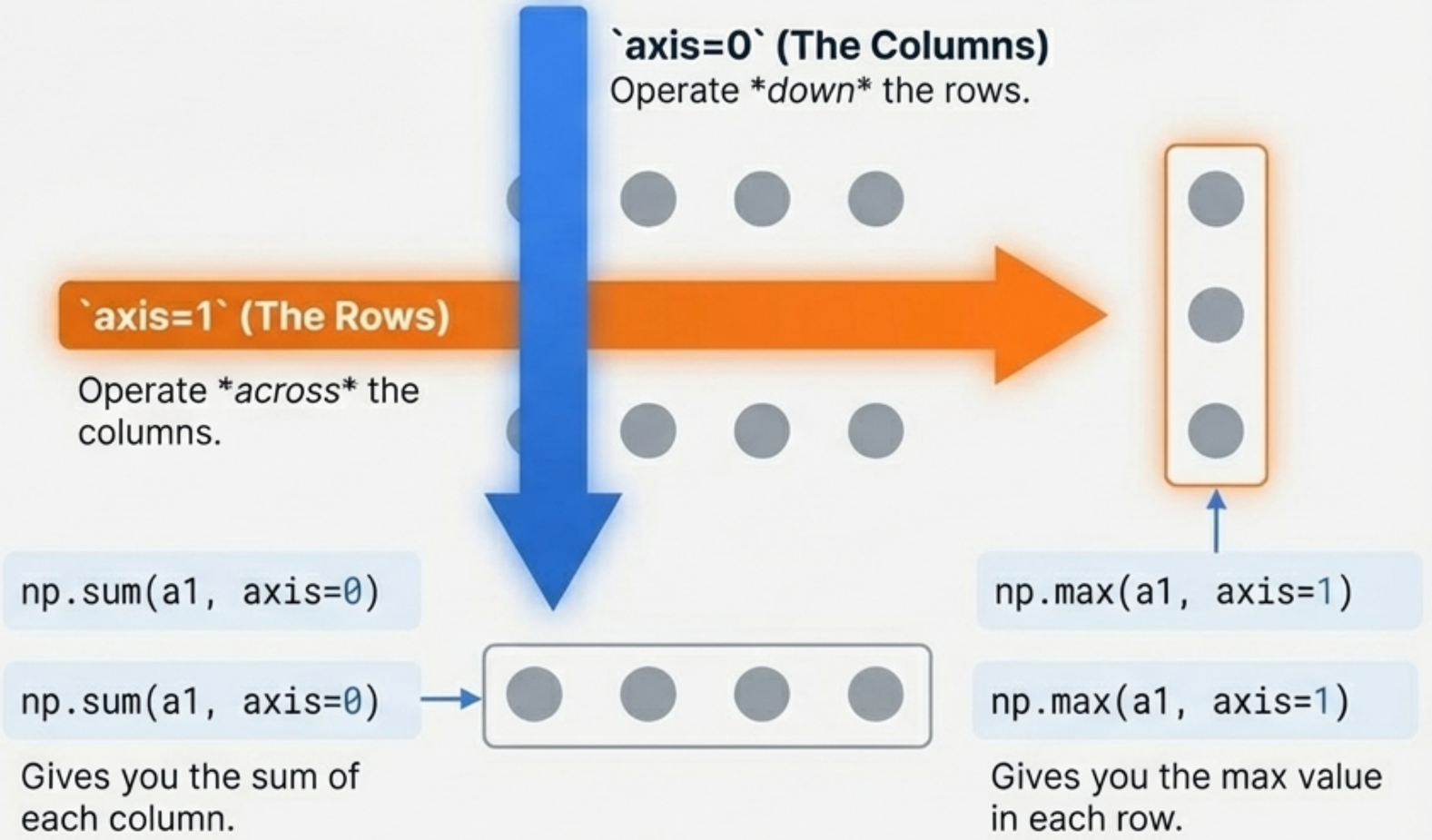
```
array([[ 6,  8],
       [10, 12]])
```

# The NumPy Function Toolkit

## Quick Stats & Aggregations

- `np.max(a1)`
  Find the max value in the entire array.
- `np.min(a1)`
  Find the minimum value.
- `np.sum(a1)`
  Sum all elements.
- `np.mean(a1)`
  Calculate the mean.
- `np.std(a1)`
  Calculate the standard deviation.

## The Most Important Parameter: `axis`



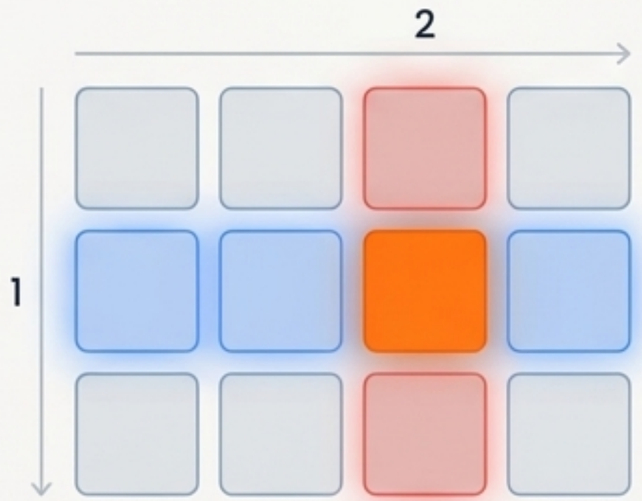
# Part 1: Indexing (Grabbing a Single Item)

### 1D Array

`a1[4]`

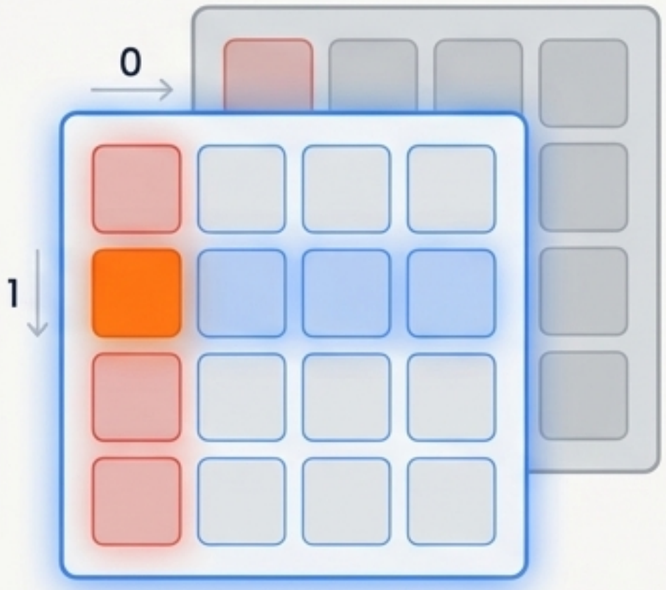

### 2D Array: `[row, column]`

`a2[1, 2]`



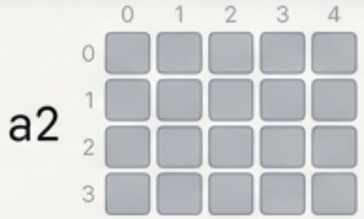

### 3D Array: `[block, row, column]`

`a3[0, 1, 0]`



# Part 2: Slicing (Grabbing Chunks of Items)

Use the colon ':' which means "from the start to the end".




## Get a full row

```
a2[1, :]
```

Get me Row 1, and all (':') of its columns.


## Get a full column

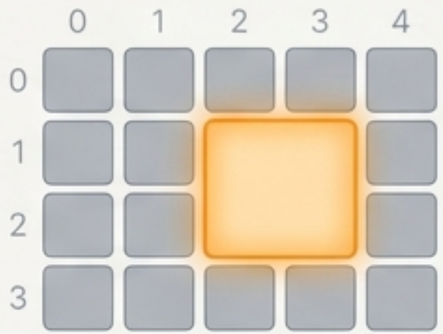
```
a2[:, 2]
```

Get me all (':') the rows, but only from Column 2.


## Get a sub-matrix

```
a2[1:3, 2:4]
```

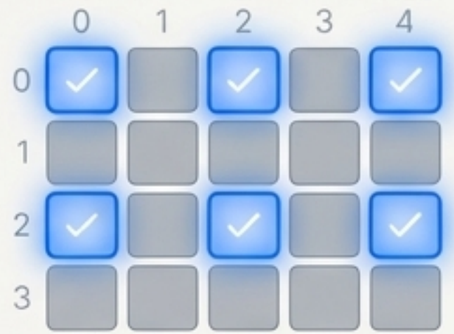
Rows 1 up to (not including) 3. Columns 2 up to (not including) 4.




## Using Steps

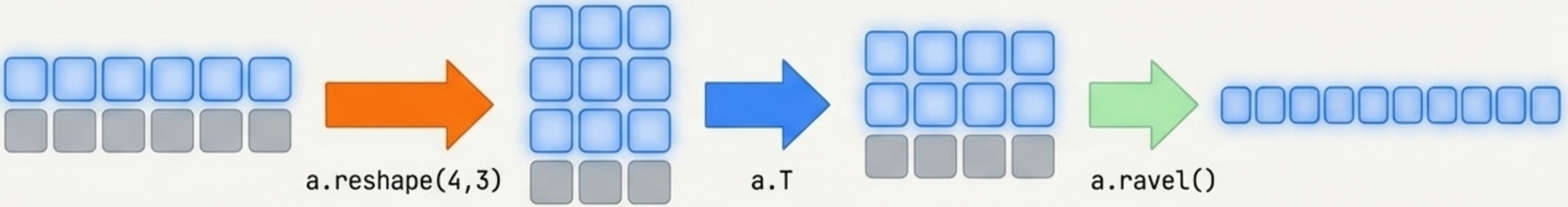
```
a2[::2, ::2]
```

From start to end, give me every 2nd row and every 2nd column.



# Reshaping Your World

Sometimes you need to change the layout of your data without changing the data itself.




Initial Array: (2, 6)

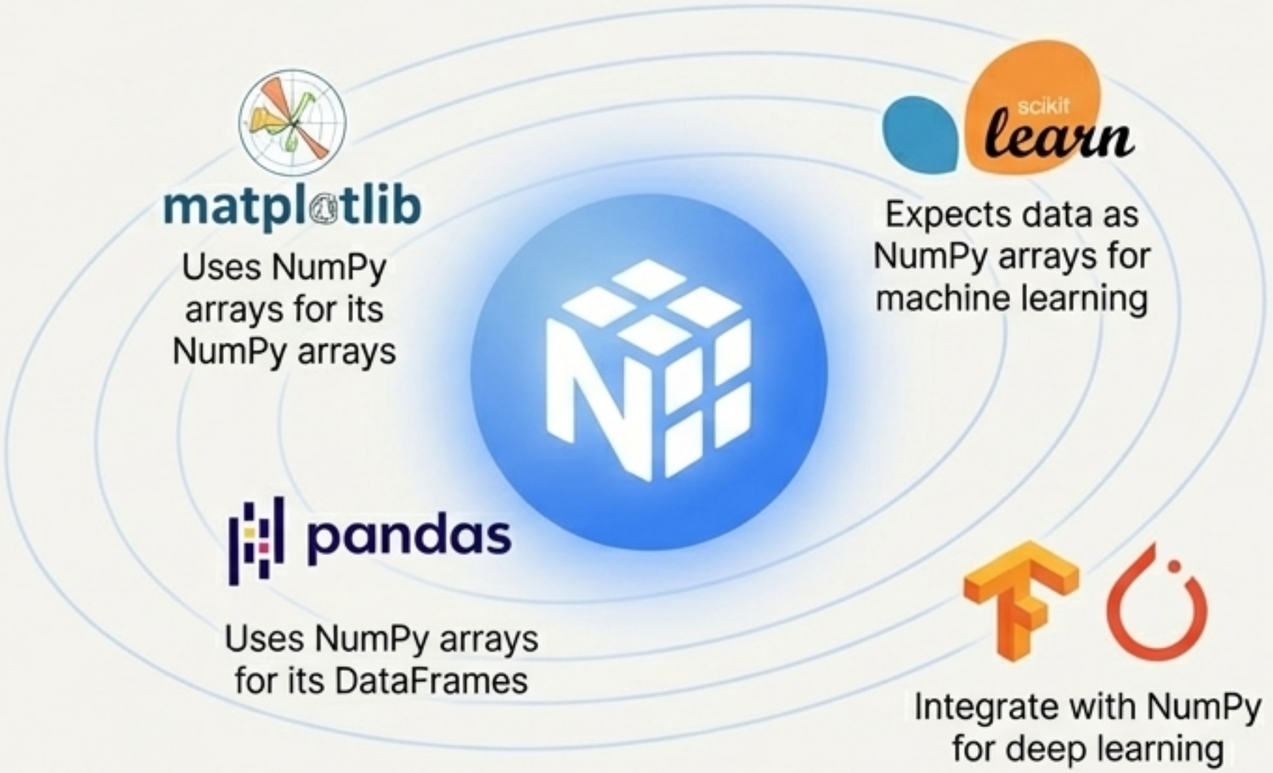
**Reshape:** Change the dimensions of an array, like a 2x6 into a 4x3. The total number of elements (**size**) must remain the same!

**Transpose:** Flip the rows and columns. A 4x3 becomes a 3x4.

**Ravel:** Flatten any array, no matter its dimension, into a single 1D row.

# The Foundation of Python's Data Ecosystem

- NumPy solves Python's speed problem with its `ndarray` object, using contiguous memory and vectorization.
- It gives you a powerful, loop-free syntax for mathematical and logical operations on entire datasets.
- It provides the fundamental array object that nearly every other major data library is built upon.




**Master NumPy, and you've mastered the building block of modern data science in Python.**