# Unlocking Efficiency

## *The Power of Vectorization*

***Milan Ondrašovič***

**PyData** - 29. 2. 2024

**Machine learning research engineer** at ROSSUM Czech Republic, s.r.o.

✉️ milan.ondrasovic@gmail.com | 📃 https://github.com/mondrasovic

# Introduction

# Prerequisites

Python

NumPy

Algebra

https://python.org | https://numpy.org

# What is meant by vectorization?

Rewriting code to utilize vector operations, enabling loop execution in languages like C/C++ or Fortran to leverage SIMD instructions for computational efficiency★

“ Rule of thumb: "***Avoid loops unless necessary!***" ”

★It's not **image conversion** (bitmap to vector representation) or a **matrix transformation** (matrix to column vector)

# Motivating Example

# Vectorization Demo - Task Definition

Compute a sum of squares of multiple numbers

Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ be a list of $n$ real numbers. The *sum of squares* is defined as

$$s = \sum_{i=1}^{n} x_i^2$$

# Pure Python Implementation

```python
def calc_squared_sum_basic(vals):
    result = 0
    for val in vals:
        result += val ** 2
    return result
```

```python
def calc_squared_sum_comprehension(vals):
    return sum(val ** 2 for val in vals)
```

> Use comprehensions if appropriate

Comprehensions are **concise** and often **faster** (with exceptions... like this one 😄)

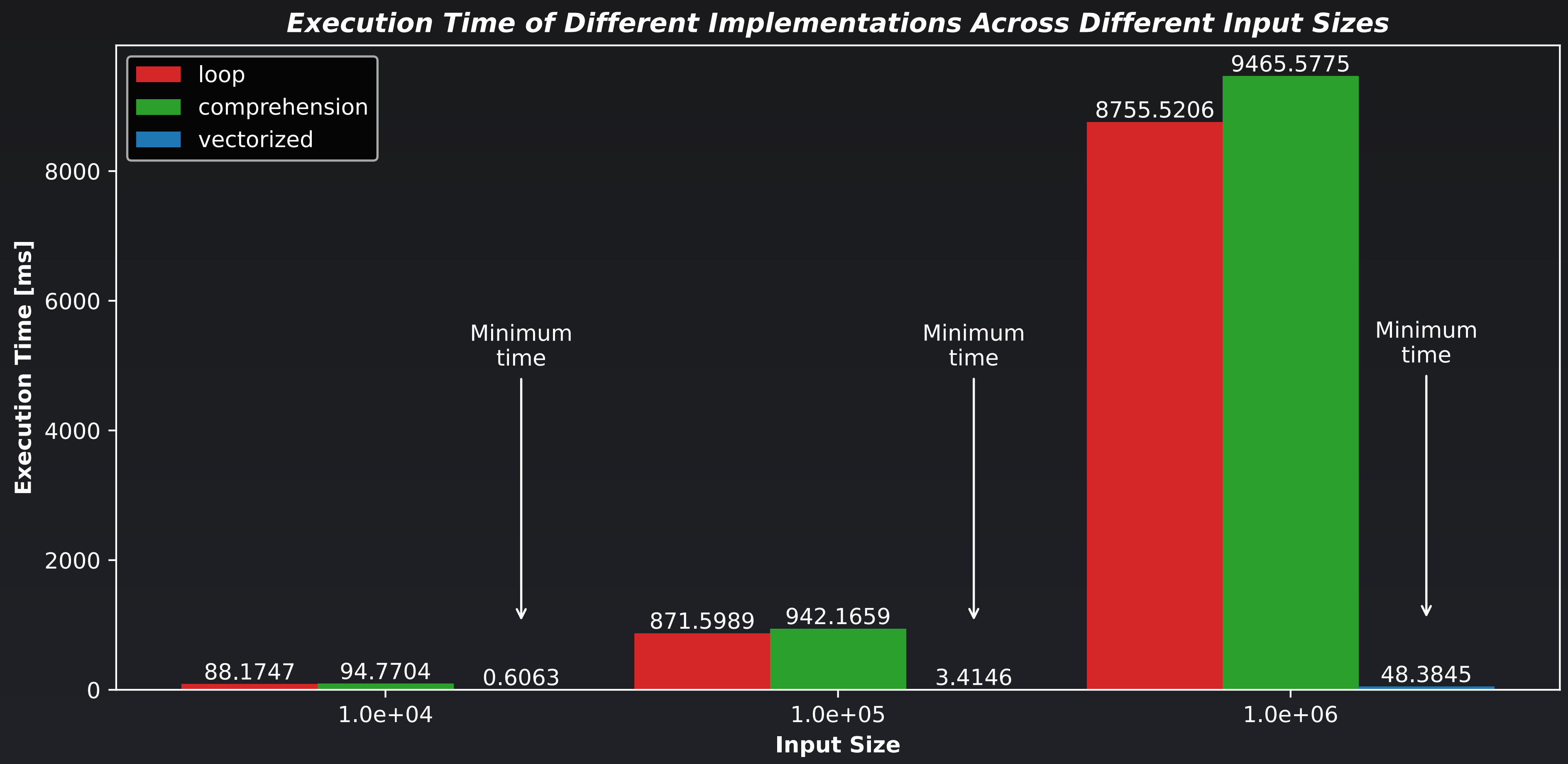
# Vectorized Implementation

```python
import numpy as np

def calc_squared_sum_vectorized(vals):
    return np.sum(np.asarray(vals) ** 2)
```

# Benchmarking - Statistics

**Minimum time** in **seconds** from 10 arithmetic means of 100 function executions




Hardware Specification: CPU: AMD Ryzen™ 5 5500, 6 cores/12 threads, Max. boost clock 4.2GHz, L2 cache 3MB

# Common Use Cases

# Adoption Among Various Libraries

- ⚖️ Similar approach to vectorization "syntax", e.g.:
  - NumPy
  - PyTorch
  - TensorFlow
- 🏛️ Many libraries are built on top of NumPy, e.g.:
  - opencv-python
  - scikit-learn
  - scikit-image

There are **countless of libraries** already encompassing plethora of applications

# DSL-Like Aspects

🤔

Writing vectorized code feels like working with a domain-specific language (DSL)★

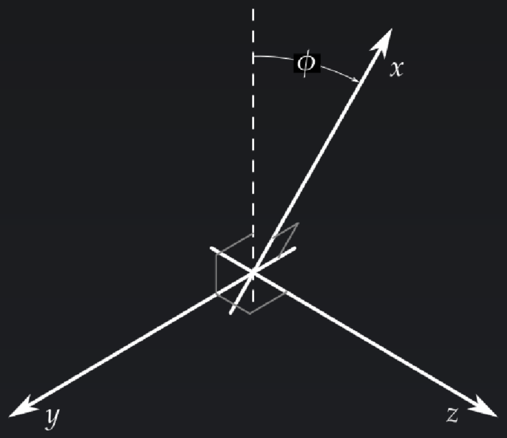
★This is just a **personal feeling**, not a general statement

# Properties Resembling a Domain-Specific Language (DSL)

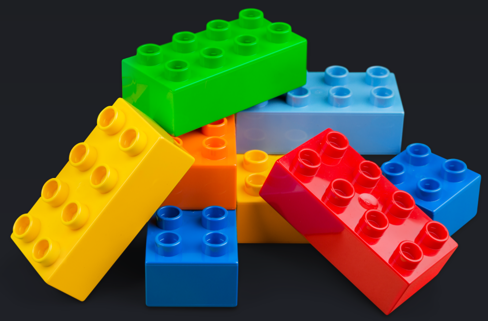
**Conciseness and expresiveness**

**Shift in perspective**

**"Standard" tooling**




- High-level abstractions
- Array-oriented thinking
- Library functions and built-in operators



# Use Cases for Various Roles

💻 Programmers

📊 Data scientists

🧠 ML engineers

- Numerical computing
- Data preprocessing and *Exploratory Data Analysis*
- Machine learning model training and inference

This list of **roles** and **applications** is just illustrative, not exhaustive 🙂

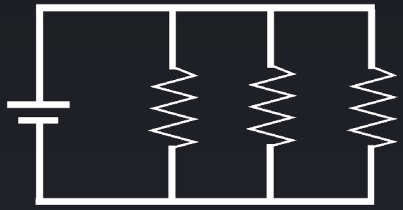
# Vectorization Impacts

# Impacts of Vectorization On Code

- Execution
- Implementation
- Codebase

# Execution Speed and Loading Time


- 👍 Even 1000x faster
- 👍 Parallel instruction execution
- 👎 Slower if used improperly
- 👎 Longer loading times (heavy imports)



These statements **generally hold true**, but sometimes they don't (**exceptions exist**) 🤷

# Transparency

👍

In some libraries, transparent execution on both CPU and GPU

👎

Sometimes, need to transfer data between RAM and GPU RAM

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
cpu_tensor = torch.randn(3, 3)
gpu_tensor = cpu_tensor.to(device)
```

For certain **Mac GPU** 🍎 devices, there is `torch.device("mps")` (*Metal Performance Shaders*) | https://pytorch.org/

# Data Types

👍

Internal data type-specific optimizations

👎

Automatic conversion pitfalls

```
>>> py_list = ["ab", "cde"]
>>> py_list[0] = "fghi"
>>> py_list  # The first element is completely replaced
['fghi', 'cde']
>>> np_array = np.asarray(["ab", "cde"])
>>> np_array[0] = "fghi"
>>> np_array  # The string is truncated - max. length is 3
array(['fgh', 'cde'], dtype='<U3')
```

Underlying **optimizations** with respect to data types might violate some **Pythonic assumptions** 😱

# Code Behavior and Error Handling

- 👍 Expected behavior across libraries
- 👍 Less error-prone code
- 👎 Certain assumptions may mislead
- 👎 Internal errors (C/C++) are "unreadable"

> To paraphrase *Donald Knuth*:
> "Built-in optimization *may* be the root of *some* evil"

Example on the next slide ➡️

# Views Vs. Original Arrays

```
>>> py_list = [1, 2, 3, 4]
>>> py_list_new = py_list[:2]  # A new copy is created
>>> py_list_new[0] = 5
>>> py_list_new
[5, 2]
>>> py_list  # No modification to the original data
[1, 2, 3, 4]

>>> np_array = np.asarray([1, 2, 3, 4])
>>> np_array_new = np_array[:2]  # Just a view is created
>>> np_array_new[0] = 5
>>> np_array_new
array([5, 2])
>>> np_array  # The original array has been modified
array([5, 2, 3, 4])
```

It's better to **assume** that you are working with a **view**, unless proven otherwise 😱

# Maintenance and Portability

- 👍 Outsourcing computation to an external library
- 👍 Often painless porting between libraries
- 👎 Dependency on an external library
- 👎 Some operations are not available in all libraries

> There is a fine line between vectorizing and obfuscating

In this regard, if something goes wrong, then it does so spectacularly! 😆

# Readability and Onboarding

- 👍 Brevity can be a blessing
- 👍 Easier knowledge transfer
- 👎 Brevity can be a curse
- 👎 A priori knowledge of vectorization required

> Even a seasoned programmer may feel like a newbie

# Broadcasting

# Broadcasting

The smaller array is “broadcast” across the larger array to assure shape compatibility

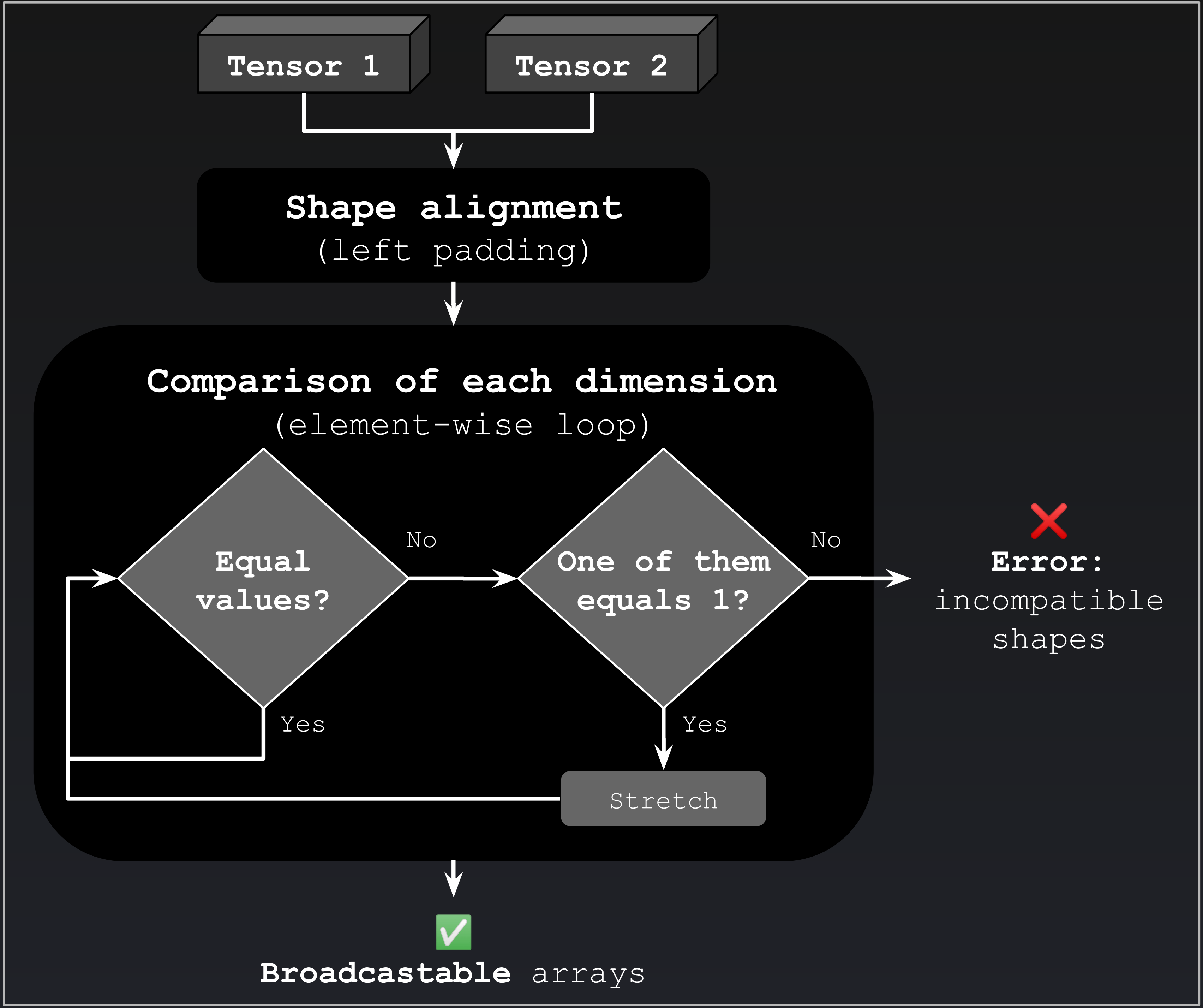
The `np.broadcast_to` function "simulates" the effect

**Fun Fact**

Even some mathematical books★ have adopted the *broadcasting notation* to simplify formulas

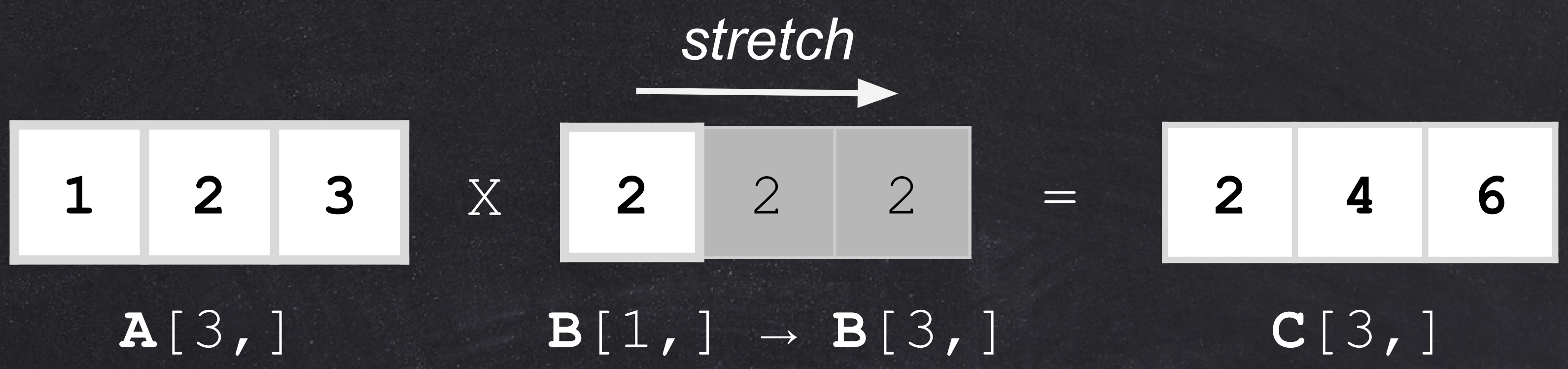
★For example, the book "**Deep Learning**" by I. Goodfellow, Y. Bengio, and A. Courville

# Rules


Tensor 1
Tensor 2
Shape alignment
(left padding)
Comparison of each dimension
(element-wise loop)
Equal values?
No
One of them equals 1?
No
Error:
incompatible shapes
Yes
Yes
Stretch
Broadcastable arrays

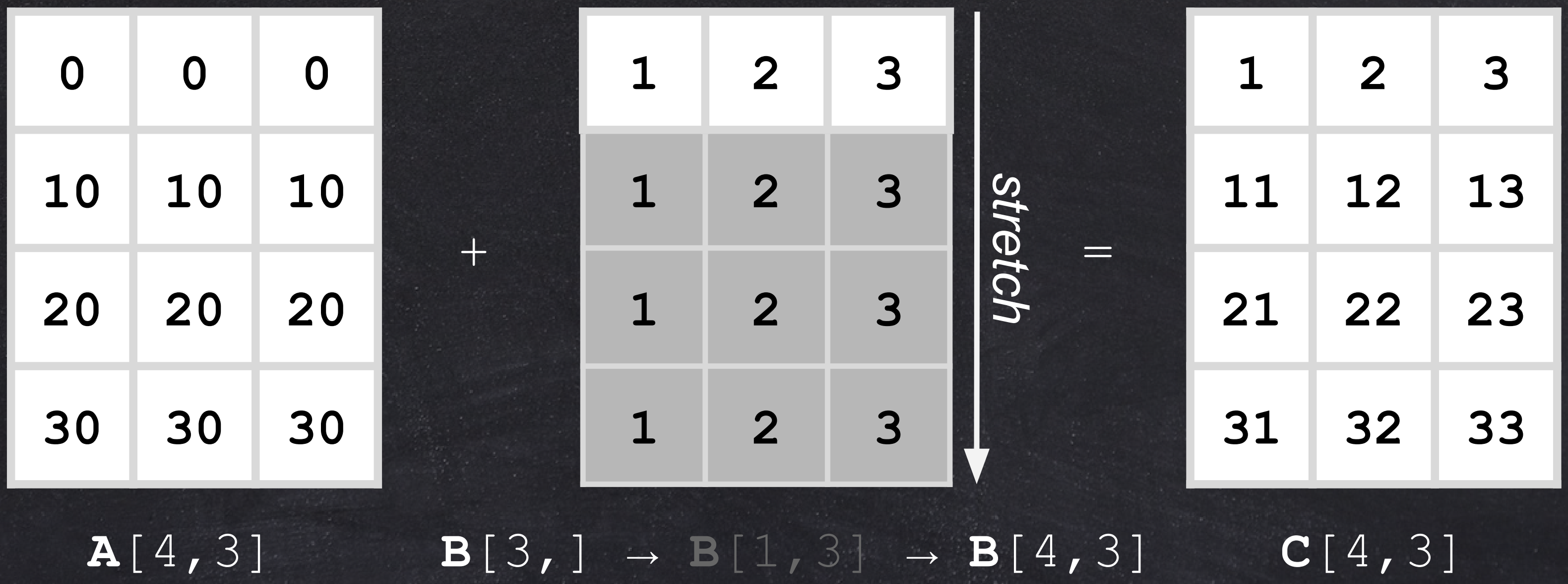
# Broadcasting - Single Array (1D)

```
>>> import numpy as np
>>> a = np.asarray([1, 2, 3])  # Shape: (3,)
>>> b = 2  # Equivalents: np.asarray(2), [2], (2,)
>>> a * b  # (3,) * () | (3,) * (1,) | (3,) * (3,)
array([2, 4, 6])
```



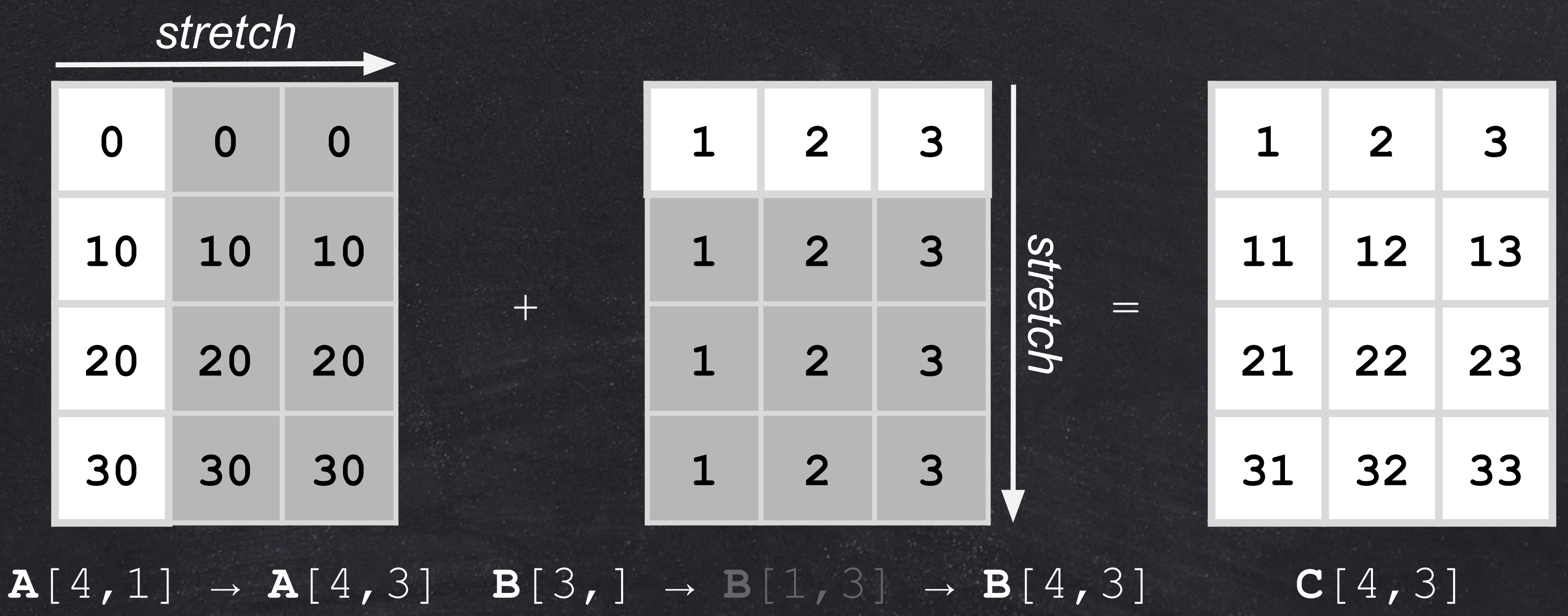
# Broadcasting - Single Array (2D)

```
>>> import numpy as np
>>> a = np.asarray([[0, 0, 0], [10, 10, 10], [20, 20, 20], [30, 30, 30]])  # Shape: (4, 3)
>>> b = np.asarray([1, 2, 3])  # Shape: (3,)
>>> a + b  # (4, 3) + (3,) | (4, 3) + (1, 3) | (4, 3) + (4, 3)
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```



# Broadcasting - Both Arrays (2D)

```
>>> import numpy as np
>>> a = np.asarray([0, 10, 20, 30])[..., np.newaxis]  # Shape: (4, 1)
>>> b = np.asarray([1, 2, 3])  # Shape: (3,)
>>> a + b  # (4, 1) + (3,) | (4, 1) + (1, 3) | (4, 3) + (1, 3) | (4, 3) + (4, 3)
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```



# Einstein Notation

# Einstein Notation

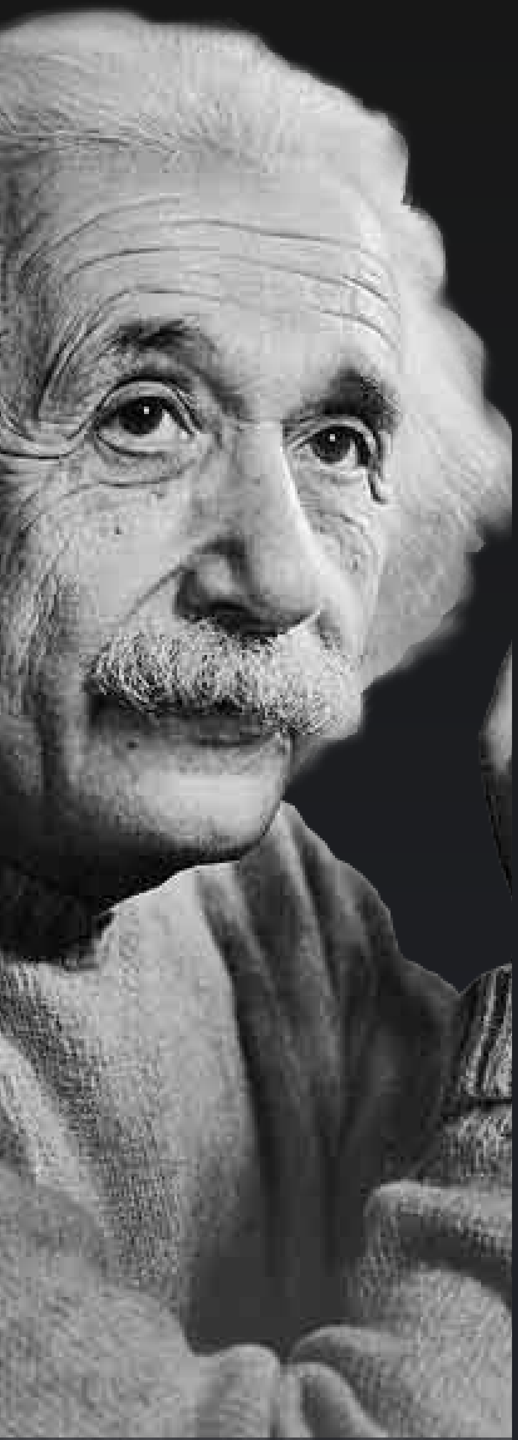
Notational convention that implies summation over a set of indexed terms, thus achieving brevity

$$y = \sum_{i=1}^{3} c_i \mathbf{x}^i = c_1 \mathbf{x}^1 + c_2 \mathbf{x}^2 + c_3 \mathbf{x}^3$$

is simplified into

$$y = c_i \mathbf{x}^i$$

https://en.wikipedia.org/wiki/Einstein_notation

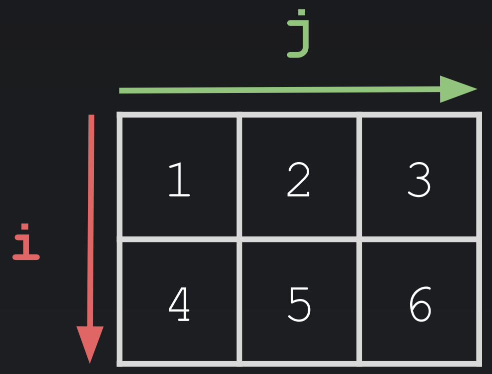


# Einstein Summation

Concise and efficient method for various tensor operations utilizing string notation to specify indexing

> Advice: If possible, standard functions should be preferred to `np.einsum`

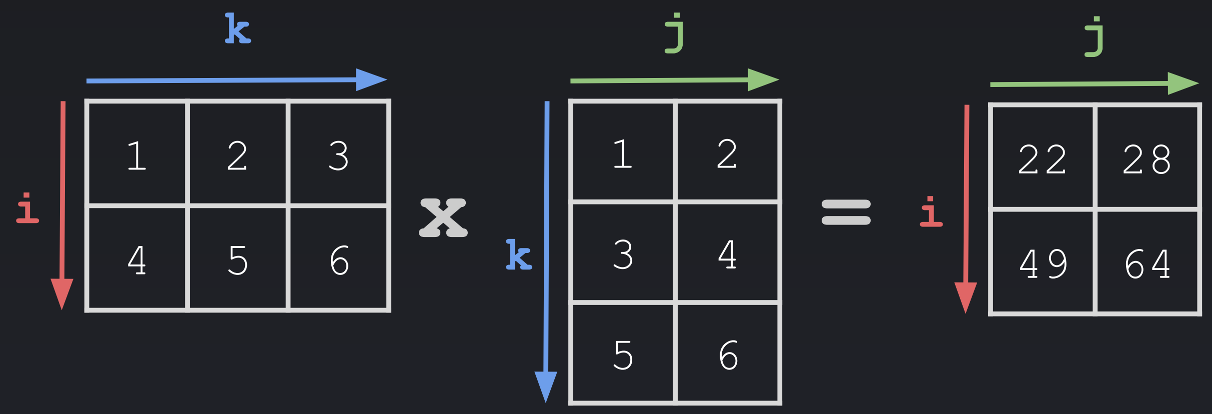
# Element Sum (Example)



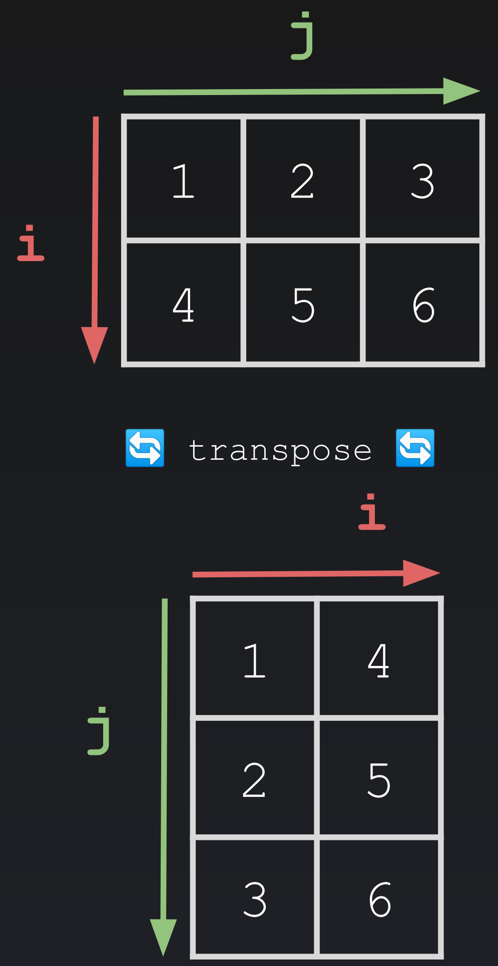

```
>>> import numpy as np
>>> arr = np.asarray([[1, 2, 3], [4, 5, 6]])
>>> np.einsum("ij ->", arr)
21
>>> np.sum(arr)
21
```

# Matrix Multiplication (Example)

```
>>> import numpy as np
>>> a = np.asarray([[1, 2, 3], [4, 5, 6]])
>>> b = np.asarray([[1, 2], [3, 4], [5, 6]])
>>> np.einsum("ik, kj -> ij", a, b)
array([[22, 28],
       [49, 64]])
>>> np.matmul(a, b)  # Or equivalently `a @ b`
array([[22, 28],
       [49, 64]])
```



# Transpose of a Matrix (Example)




```
>>> import numpy as np
>>> arr = np.asarray([[1, 2, 3], [4, 5, 6]])
>>> np.einsum("ji", arr)  # Notice the order of indices
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.transpose(arr)  # Or equivalently `arr.T`
array([[1, 4],
       [2, 5],
       [3, 6]])
```

For **arbitrary swapping of axes**, see `np.swapaxes`

# Einstein "Operations" Notation

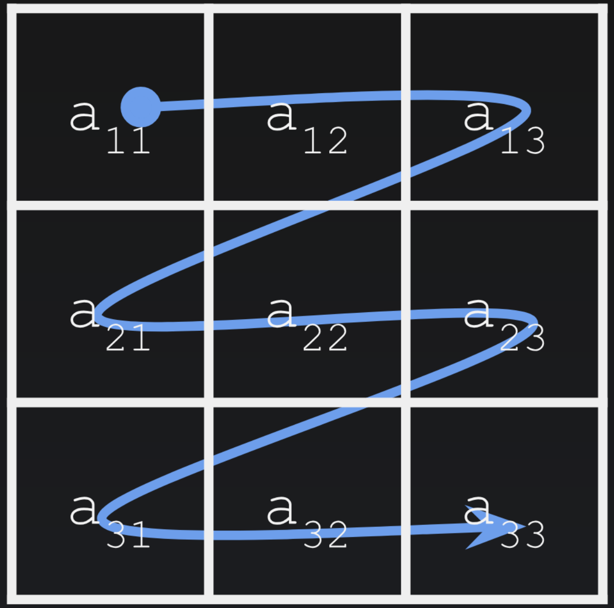
String-based notation (similar to Einstein's) to perform various tensor operations 🤸

```
>>> import numpy as np
>>> from einops import rearrange, reduce, repeat
>>> img = np.random.random((128, 256, 3))  # Shape: (height, width, n_channels)
>>> rearrange(img, "height width n_channels -> n_channels height width").shape
(3, 128, 256)
>>> reduce(img, "height width n_channels -> width height", "max").shape
(256, 128)
>>> repeat(img, "height width n_channels -> height (tile width) n_channels", tile=2).shape
(128, 512, 3)
```

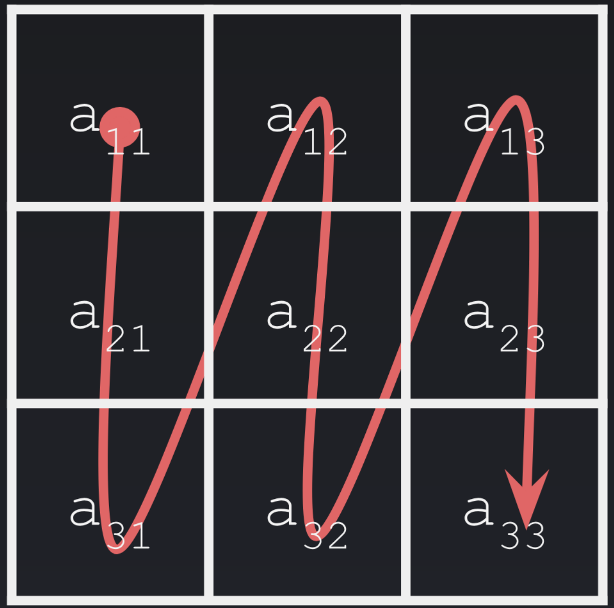
Supports **NumPy**, **PyTorch**, **TensorFlow**, **JAX**, and others 😍 | https://pypi.org/project/einops/

# Other Performance Tips

***Row-major*** **order**




***Column-major*** **order**




# Row-major vs. Column-major

C-contiguous: row-major (default)

Fortran-contiguous: column-major

```
>>> import numpy as np
>>> np.arange(6).reshape(2, 3, order="C")
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.arange(6).reshape(2, 3, order="F")
array([[0, 2, 4],
       [1, 3, 5]])
```

Beware of memory **spatial** and **temporal locality** - big arrays, big difference!🤔

# Modifying Arrays

Arrays allocated as a contiguous block of memory

**Avoid resizing memory** in a loop - if needed, **pre-allocate** it using `np.empty` and then fill in place

| | | Memory order | |
|---|---|---|---|
| | | *Row-major* | *Column-major* |
| **Stacking direction** | *Vertical* | ❎ | ❌ |
| | *Horizontal* | ❌ | ❎ |

💡 `np.hstack`, `np.vstack`, and `np.stack` use `np.concatenate` internally - **axis** and **memory order** impact the speed

# View vs. Copy

```python
import numpy as np
arr = np.asarray([1, 2, 3, 4, 5, 6])  # `np.asarray` does not copy, `np.array` does
```

```python
sliced_arr = arr[1:4]  # VIEW
reshaped_arr = arr.reshape(2, -1)  # VIEW | Shape: (2, 3)
transposed_arr = arr[np.newaxis].T  # VIEW | Shape: (6, 1)
raveled_arr = orig_arr.ravel()  # VIEW | Shape: (6,)
```

```python
mask_selected_arr = arr[[False, True, True, True, False, False]]  # COPY
fancy_indexed_arr = arr[[1, 2, 3]]  # COPY
flattened_arr = orig_arr.flatten()  # COPY | Shape: (6,)
```

💡Use `new_arr.base is orig_arr` to test whether `new_arr` is just a **view** of `orig_arr` and not a new **copy**

# Unbuffered In-place Operations

Family of *universal functions*★: `np.ufunc.at`, `np.ufunc.reduce`, `np.ufunc.accumulate`, etc.

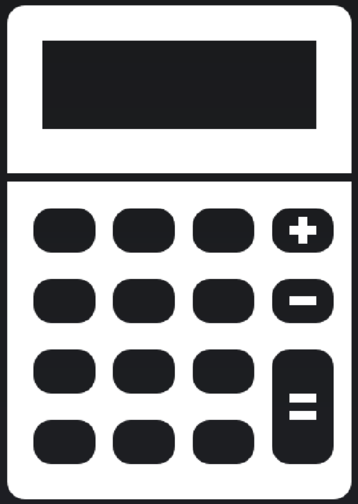
```
>>> import numpy as np
>>> arr_buffered = np.asarray([1, 2, 3, 4, 5])
>>> arr_buffered[[0, 0, 0]] += 1  # BUFFERING increments only once (keeps track)
>>> arr_buffered
array([2, 2, 3, 4, 5])

>>> arr_unbuffered = np.asarray([1, 2, 3, 4, 5])
>>> np.add.at(arr_unbuffered, [0, 0, 0], 1)  # No BUFFERING increments 3 times
>>> arr_unbuffered
array([4, 2, 3, 4, 5])  # First element: 1 + (3 * 1) = 4
```

💡★**Universal function** (`ufunc`) - operates on `ndarrays` in an **element-by-element** fashion

# Prefer Numerically Stable Functions

| Overflow | Underflow |
|---|---|



- $e^x - 1$
  - `np.exp(x) - 1` ⚠️, `np.expm1(x)` ✅
- $\log(1 + x)$
  - `np.log(1 + x)` ⚠️, `np.logp1(x)` ✅
- and *many others*... 🙄

💡Some mathematical operations have their **numerically stable counterparts** - look for them

# Making Vectorization Easier

# Harnessing `jupyter` 's Power

⌨️

Interactive mode - data inspection

⏲️

Visualization - plotting

👀

Data shape exploration

Expedite your implementation by **on-the-fly interaction with code**

# Map/Reduce Patterns

| Map (dimension preserving) | Reduce (dimension reducing) |
| --- | --- |

```
>>> import numpy as np
>>> vals = np.asarray([[1, 2, 3], [4, 5, 6]])  # Shape: (2, 3)
>>> np.power(vals, 2)  # MAPPING - preserves shape (2, 3)
array([[ 1,  4,  9],
       [16, 25, 36]])
>>> np.sum(vals, axis=1)  # REDUCTION - removes an axis, shape (2,)
array([ 6, 15])
```

**Reduction** happens along an `axis` parameter ( `keepdims=True` possible)

Array operations can generally be grouped into "**mapping**" and "**reducing**" | https://en.wikipedia.org/wiki/MapReduce

# Commenting Tensor Shapes

- Consider adding shapes as a comment
  - single-letter: `# (R, C)`
  - variable name (preferred): `# (n_rows, n_cols)`

```
def softmax(logits: np.ndarray) -> np.ndarray:
    """Transform `logits` of shape `(n_rows, n_cols)` using softmax."""
    scores = np.exp(logits)  # (n_rows, n_cols)
    row_sums = np.sum(scores, axis=1, keepdims=True)  # (n_rows, 1)
    eps = np.finfo(np.float32).eps
    probabilities = scores / (row_sums + eps)  # (n_rows, n_cols)
    return probabilities
```

Always **keep your documentation and comments up to date** - if they lie, it is worse than nothing! ⚠

# Using Advanced Type Hints

```
from typing import Annotated, Literal, TypeVar
import numpy as np
import numpy.typing as npt

DType = TypeVar("DType", bound=np.generic)

Array4 = Annotated[npt.NDArray[DType], Literal[4]]
Array3x3 = Annotated[npt.NDArray[DType], Literal[3, 3]]
ArrayNxNx3 = Annotated[npt.NDArray[DType], Literal["N", "N", 3]]
```

💡Not very common approach, just be aware of its existence

# Conclusion

# Vectorization - Conclusion

- ⏱️ Useful for **performance-demanding** applications
  - 🔃 **Looping** is performed in **low-level** languages
- 👍 Potential to vastly **improve the solution**
- 🧠 **Educational value** - learn the basic principles
  - 😱 Beware of **underlying differences**
- 🏁 It has become an **industry "standard"**
- 📚 Plenty of **resources**, solid **community support**

> ❗ It is just a tool, so don't eat soup with a fork ❗