NumPy

NumPy – huh, yeah – what’s it good for?

- NumPy introduces a new data structure: **the array**

An array is a regular, N-dimensional grid of data of the same type, typically numerical data

- Great for storing homogeneous data, where every element in the array has the same meaning. E.g. images, sound, time series

Efficient machine-native implementation

- Data is stored in a contiguous chunk of memory, using machine-native data types
- Separate metadata tells numpy how to interpret that memory as an array

Memory block storage

0	1	2	3	4	5	6	7	8

int64
8 bytes

NumPy array metadata

`dtype`	`int64`
`ndim`	`2`
`shape`	`(3, 3)`
`strides`	`(24, 8)`

NumPy view

0	1	2
3	4	5
6	7	8

Is NumPy any better than a list-of-lists?

Is NumPy any better than a list-of-lists?

- The machine-nativeness of the data storage means that common operations and algorithms can be implemented in C and Fortran, making them faster than a Python list-of- lists

- Faster in what sense?
  - Big-O complexity is the same. E.g., square matrix multiplication is still O(n^3)*
  - **It's not going to scale any better than list-of-lists**
  - **Much faster for fixed-size problems**
  - This speed advantage strictly depends on operations being made in C and Fortran. Avoid Python for-loops and use existing NumPy and SciPy functionality!

* The best matrix multiplication algorithm as of 2025 scales as O(n^2.371339). However, out-of-the-box NumPy packages usually use a Fortran library called OpenBLAS that implements a O(n^3) algorithm

Same data, different views

Memory block storage

0	1	2	3	4	5	6	7	8

int64
8 bytes

NumPy array metadata

dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

NumPy view

0	1	2
3	4	5
6	7	8

Same data, different views

Memory block storage

0	1	2	3	4	5	6	7	8

int64
8 bytes

NumPy array metadata

`dtype`	`int64`
`ndim`	`1`
`shape`	`(9,)`
`strides`	`(8,)`

NumPy view

0	1	2	3	4	5	6	7	8

O(1) operations in NumPy

When NumPy can execute a command by just changing the metadata, it does. The result is a **new view of the same data in memory**

Memory block

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

NumPy operation — **NumPy array metadata** — **NumPy view**

`x`

`ndim`	`2`
`shape`	`(4, 3)`
`strides`	`(24, 8)`

0	**1**	**2**
3	**4**	**5**
6	**7**	**8**
9	**10**	**11**

`x.ravel()`

`ndim`	`1`
`shape`	`(12,)`
`strides`	`(8,)`

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

`x.T`

`ndim`	`2`
`shape`	`(3, 4)`
`strides`	`(8, 24)`

0	**3**	**6**	**9**
1	**4**	**7**	**10**
2	**5**	**8**	**11**

`x.reshape((2, 6))`

`ndim`	`2`
`shape`	`(2, 6)`
`strides`	`(48, 8)`

0	**1**	**2**	**3**	**4**	**5**
6	**7**	**8**	**9**	**10**	**11**

The golden rule of NumPy

Operations that can be executed by only changing the metadata return a “**view “ of the original data memory block**

In all other cases, it creates a **“copy”** with a new data memory block

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

```
x
```

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

0	1	2
3	4	5
6	7	8
9	10	11

Slicing

```
x[::3, ::2]
```

dtype	
ndim	
shape	
strides	

0	2
9	11

Can this operation be done just by changing the metadata?

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

```
x
```

NumPy array metadata

`dtype`	`int64`
`ndim`	`2`
`shape`	`(4, 3)`
`strides`	`(24, 8)`

0	1	2
3	4	5
6	7	8
9	10	11

Slicing

```
x[::3, ::2]
```

`dtype`	`int64`
`ndim`	`2`
`shape`	`(2, 2)`
`strides`	`(72, 16)`

0	2
9	11

Can this operation be done just by changing the metadata?
YES!

Can it be done for all slicing operations?

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

x

NumPy array metadata

Slicing always returns
a **view** of the original array

Slicing

```
x[::3, ::2]
```

dtype	int64
ndim	2
shape	(2, 2)
strides	(72, 16)

0	**2**
9	**11**

Can this operation be done just by changing the metadata?
YES!

Can it be done for all slicing operations?
YES!

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

```
x
```

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

0	1	2
3	4	5
6	7	8
9	10	11

Fancy indexing

$$x[\underbrace{[0,\ 2,\ 3]}_{\text{row indices}},\ \underbrace{[1,\ 2,\ 1]}_{\text{column indices}}]$$

dtype	
ndim	
shape	
strides	

1	10	8

Can this operation be done just by changing the metadata?

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

NumPy array metadata

x

Fancy indexing always returns a **copy** of the original array

Fancy indexing

$$\texttt{x[}\underbrace{\texttt{[0, 2, 3]}}_{\text{row indices}}\texttt{, }\underbrace{\texttt{[1, 2, 1]}}_{\text{column indices}}\texttt{]}$$

dtype	int64
ndim	1
shape	(3,)
strides	???

1	10	8

Can this operation be done just by changing the metadata? NO!

View or copy? Quiz

Exercise

```
exercises/numpy_view_or_copy/
view_or_copy_interactive.ipynb
```

Changing one view changes them all

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

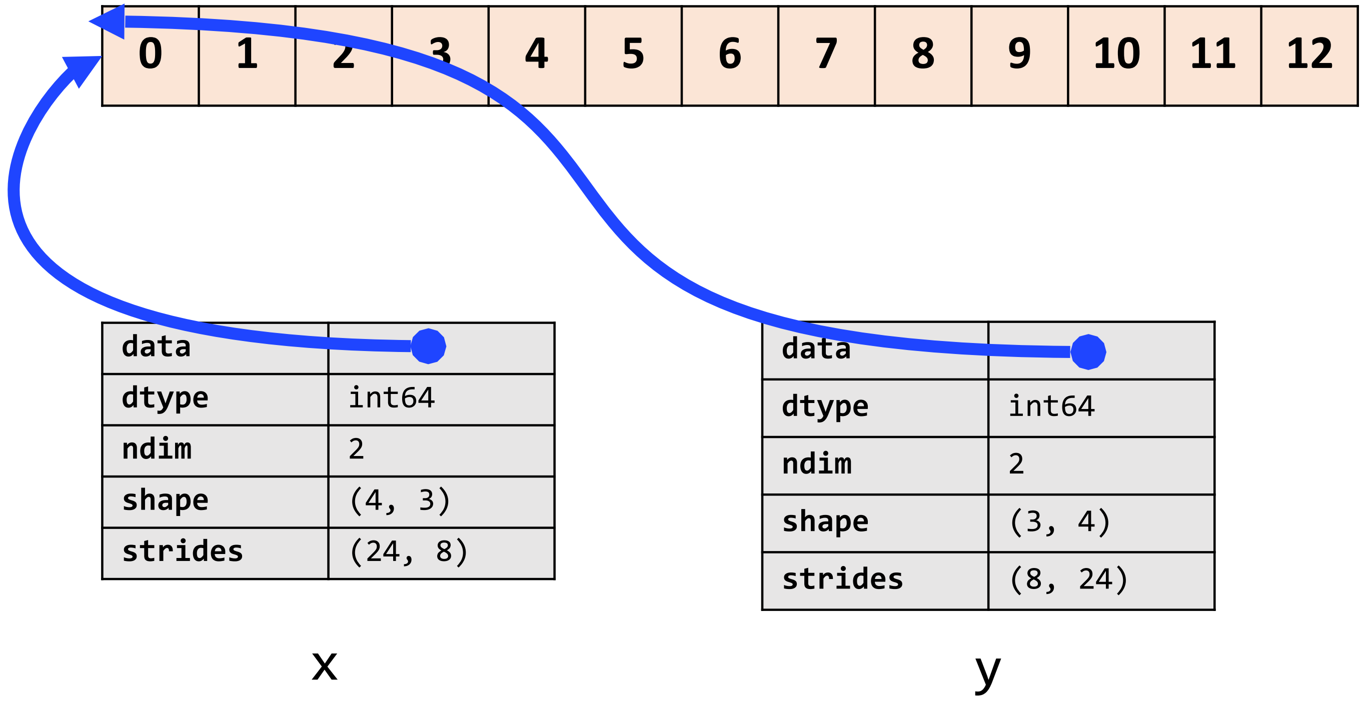
```
x
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
y = x.T
```

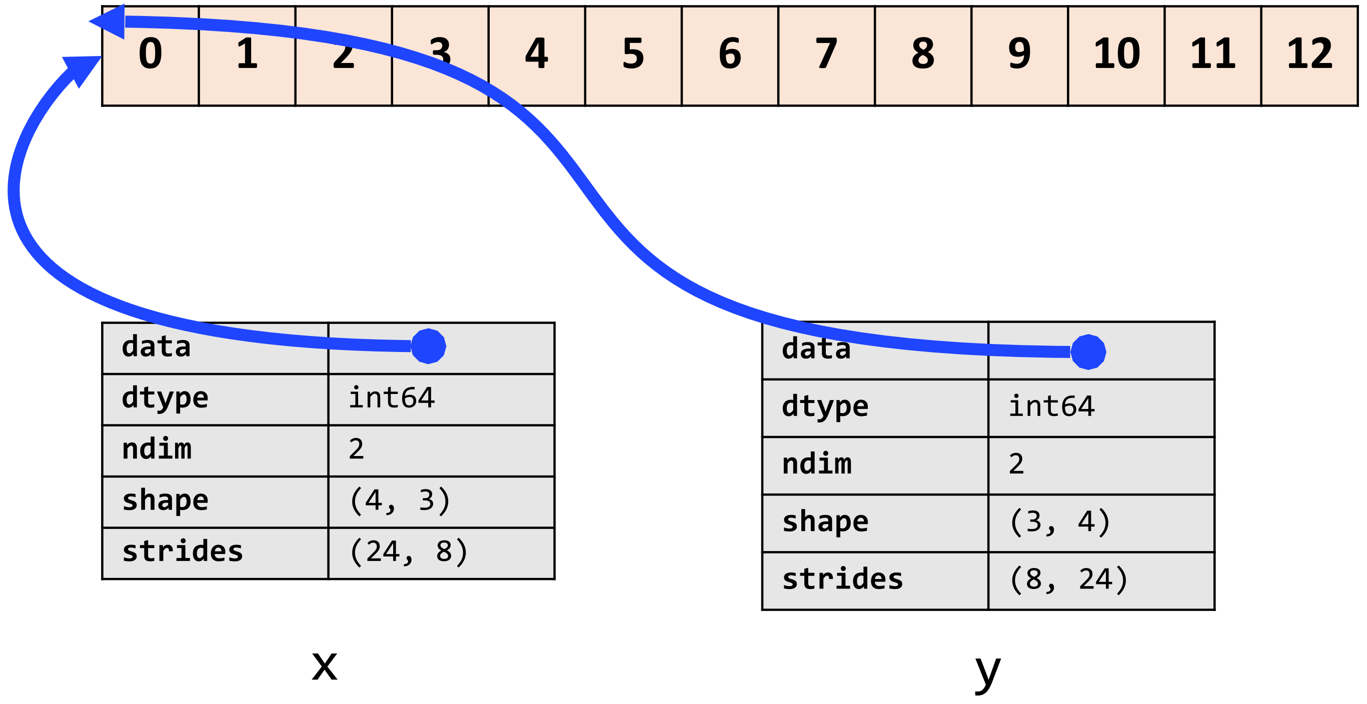
```
y
```

```
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```



Changing one view changes them all

```
y[0, 0] = 13
```



Changing one view changes them all

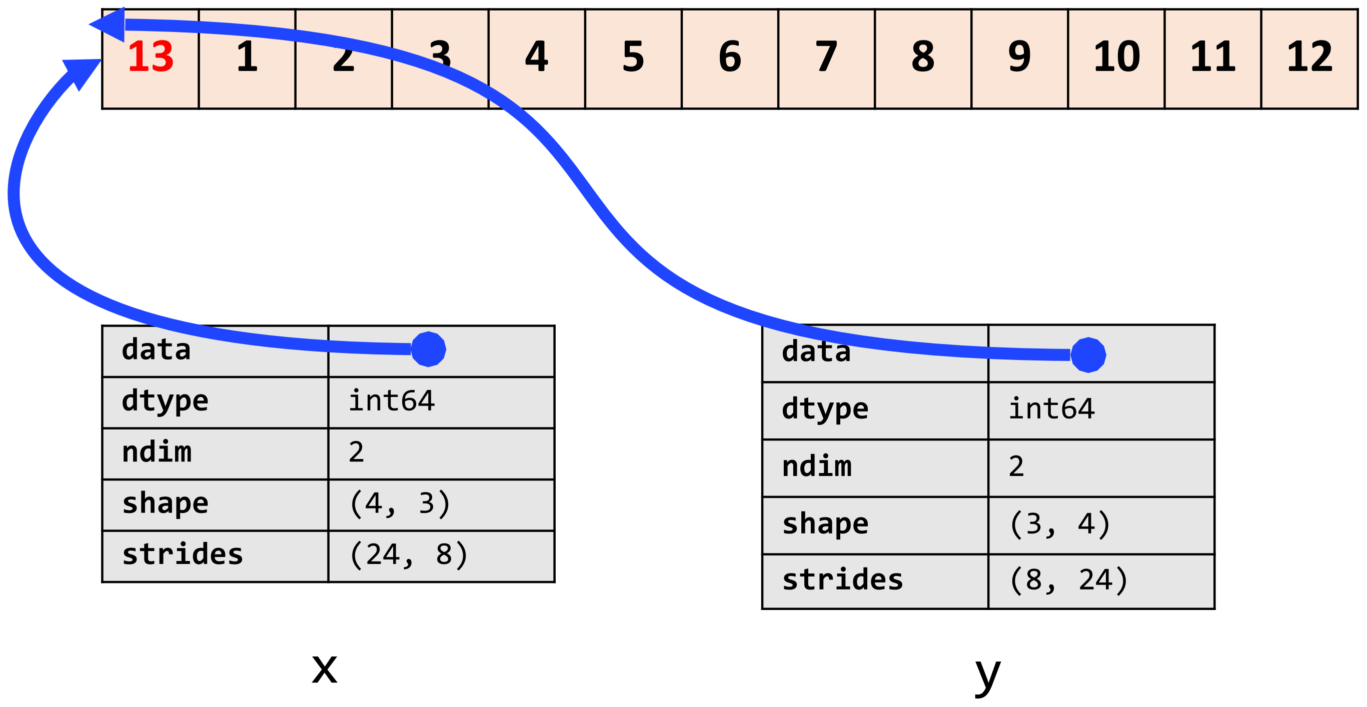
```
y[0, 0] = 13
```

```
y
```

```
array([[13,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```

```
x
```

```
array([[13,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```



data	
dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

data	
dtype	int64
ndim	2
shape	(3, 4)
strides	(8, 24)

Careful with your functions

```
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x[x == 0] = cte
    return np.log(x)
```

Careful with your functions

```python
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x[x == 0] = cte
    return np.log(x)
```

```python
a = np.array([[0.3, 0.01], [0, 1]])
print(a)
```

```
[[0.3  0.01]
 [0.   1.  ]]
```

```python
# Using the NumPy's log directly
np.log(a)
```

```
/tmp/ipykernel_50294/3282750587.py:2: RuntimeWarning: divide by zero encountered in log
  np.log(a)
```

```
array([[-1.2039728 , -4.60517019],
       [       -inf,  0.        ]])
```

```python
# Our function handles values equal zero to return a small value
robust_log(a)
```

```
array([[ -1.2039728 ,  -4.60517019],
       [-23.02585093,   0.        ]])
```

Careful with your functions

```python
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x[x == 0] = cte
    return np.log(x)
```

```python
a = np.array([[0.3, 0.01], [0, 1]])
b = a[1, :]  # A view of `a`
print(b)
```

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

```python
robust_log(b)
```

```
array([-23.02585093,   0.        ])
```

```python
b
```

```
array([1.e-10, 1.e+00])
```

The input array has been modified!

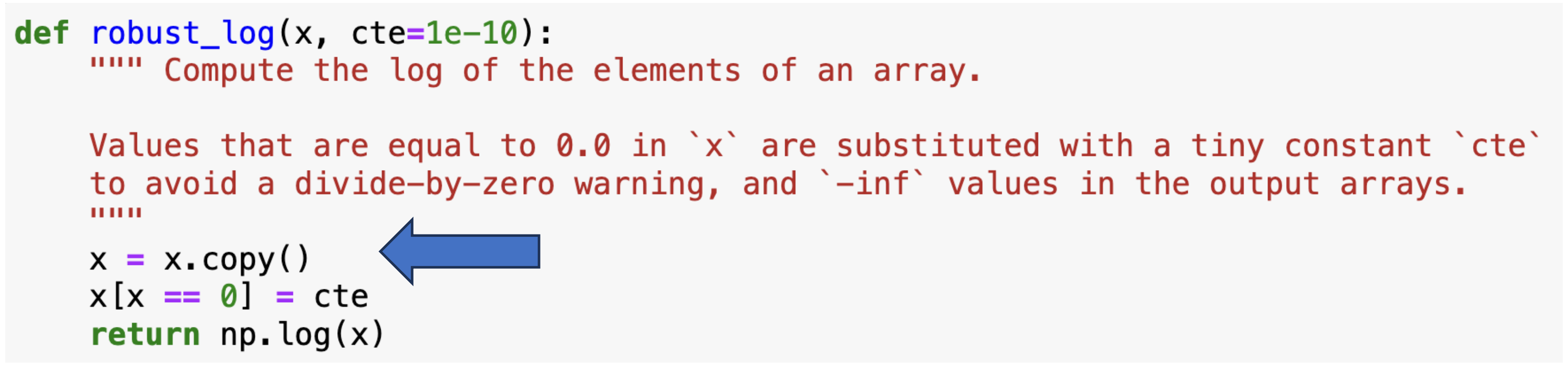
```python
a
```

```
array([[3.e-01, 1.e-02],
       [1.e-10, 1.e+00]])
```

… and so have all other views of the same data!

Careful with your functions

Best practice: functions that take an array as an input should avoid modifying it in place!
Always make a copy or be super extra clear in the docstring



```python
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x = x.copy()
    x[x == 0] = cte
    return np.log(x)
```

NumPy views and copies summary

View

- There can be multiple views of the same memory block, interpreted as different arrays
- Slicing returns a view
- In-place operations on a view modify the memory block and all of its views

Copy

- When a copy of an array needs to be created, it allocates a separate memory block and associates it with new metadata
- Fancy indexing always returns copies
- A copy can be forced with .copy()

A special kind of view: broadcasting operations

Memory block

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

NumPy array metadata

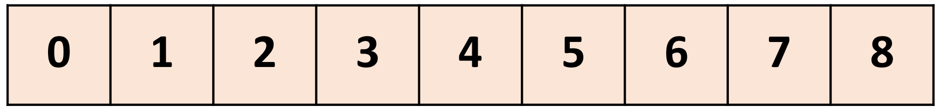
dtype	`int64`
ndim	`2`
shape	`(4, 9)`
strides	`(0, 8)`

The shape says we have 4 rows and 9 columns

A stride of 0 means that for each new row, we don't move in memory

A special kind of view: broadcasting operations

Memory block



NumPy array metadata

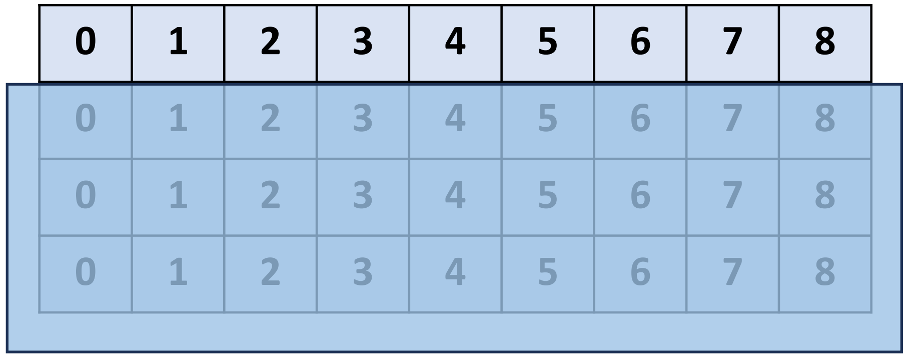
dtype	int64
ndim	2
shape	(4, 9)
strides	(0, 8)

The shape says we have 4 rows and 9 columns

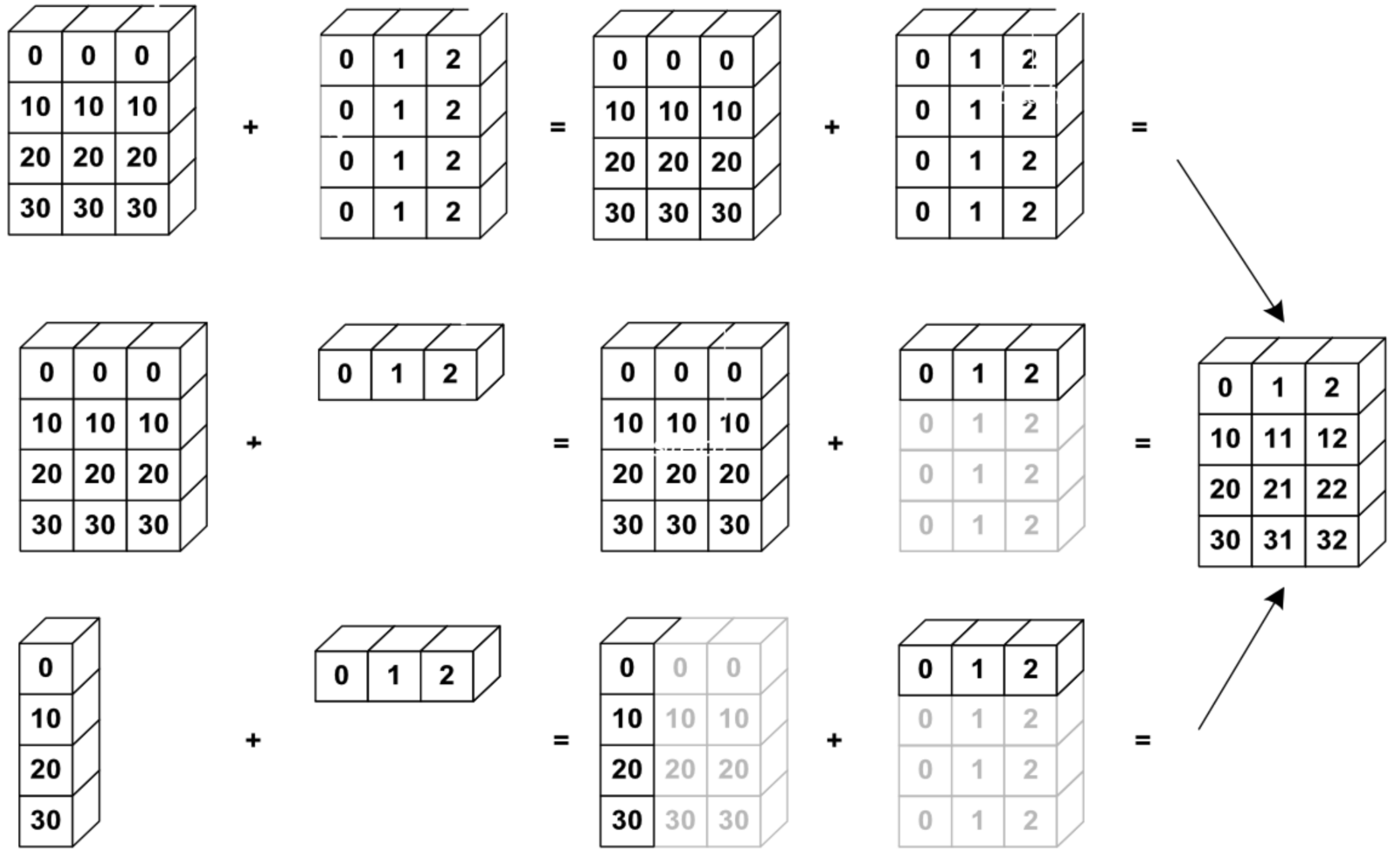
A stride of 0 means that for each new row, we don't move in memory

As a result, we obtain a view with duplicated rows, without using extra memory!

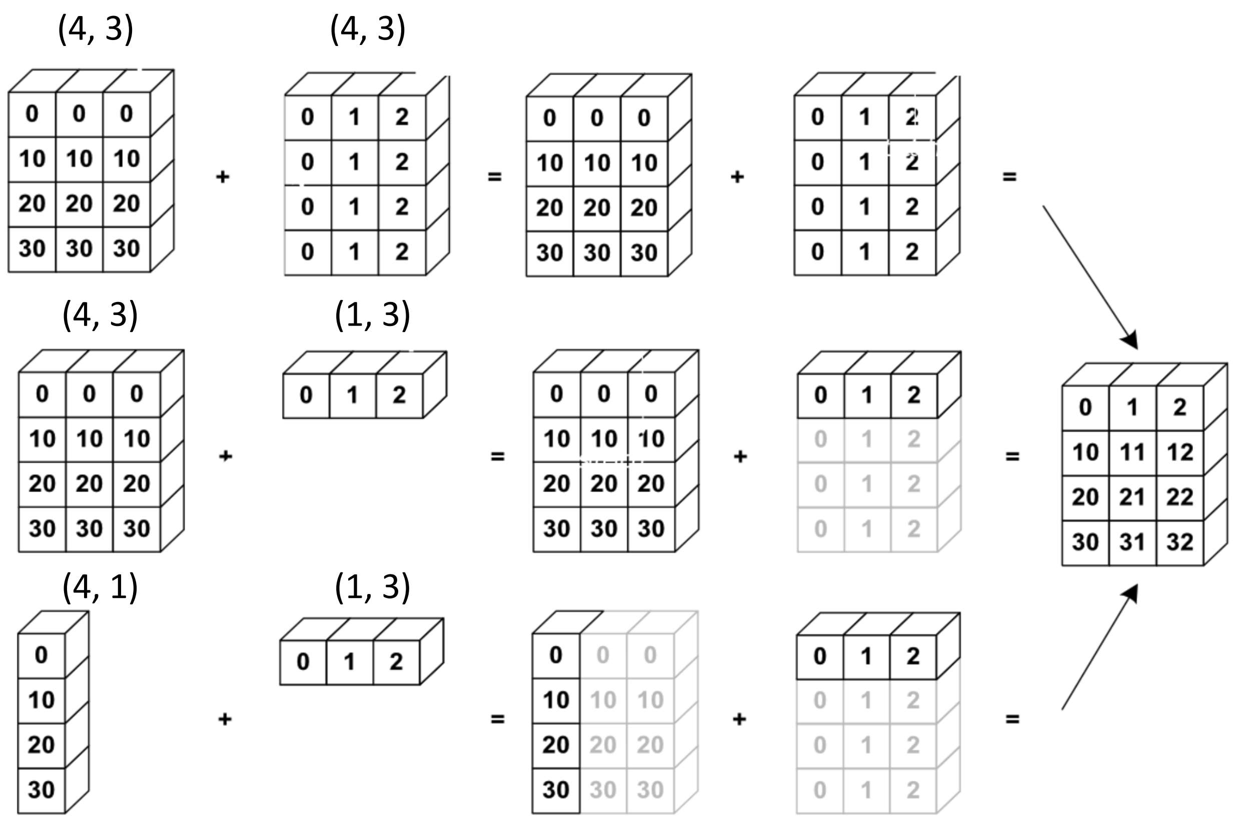
NumPy view



NumPy uses broadcasting to perform operation on arrays of different shape without having to allocate extra memory



Broadcasting matching rule: dimensions are aligned to the right and match if they are equal, or equal to 1



(4, 3) + (4, 3) -> (4, 3)

(4, 3) + (1, 3) -> (4, 3)

(4, 1) + (1, 3) -> (4, 3)

You can always add a new dimensions to make things match

```
a = np.array([0, 10, 20, 30])
b = np.array([0, 2, 3])
```

1D shape: (4,)

1D shape: (3,)

```
a + b
```

(4,) + (3,) -> ERROR!

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 a + b

ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

You can always add a new dimensions to make things match

```
a = np.array([0, 10, 20, 30])
b = np.array([0, 2, 3])
```

1D shape: (4,)

1D shape: (3,)

Add an extra dimension using np.newaxis to align them:

```
a[:, np.newaxis]
b[np.newaxis, :]
```

2D shape: (4, 1)

2D shape: (1, 3)

```
a[:, np.newaxis] + b[np.newaxis, :]
```

(**4**, **1**) + (**1**, **3**) -> (**4**, 3)

This also works (align right!):

```
a[:, np.newaxis] + b
```

(**4**, **1**) + (**3,**) -> (**4**, 3)

Broadcasting summary

Exercise

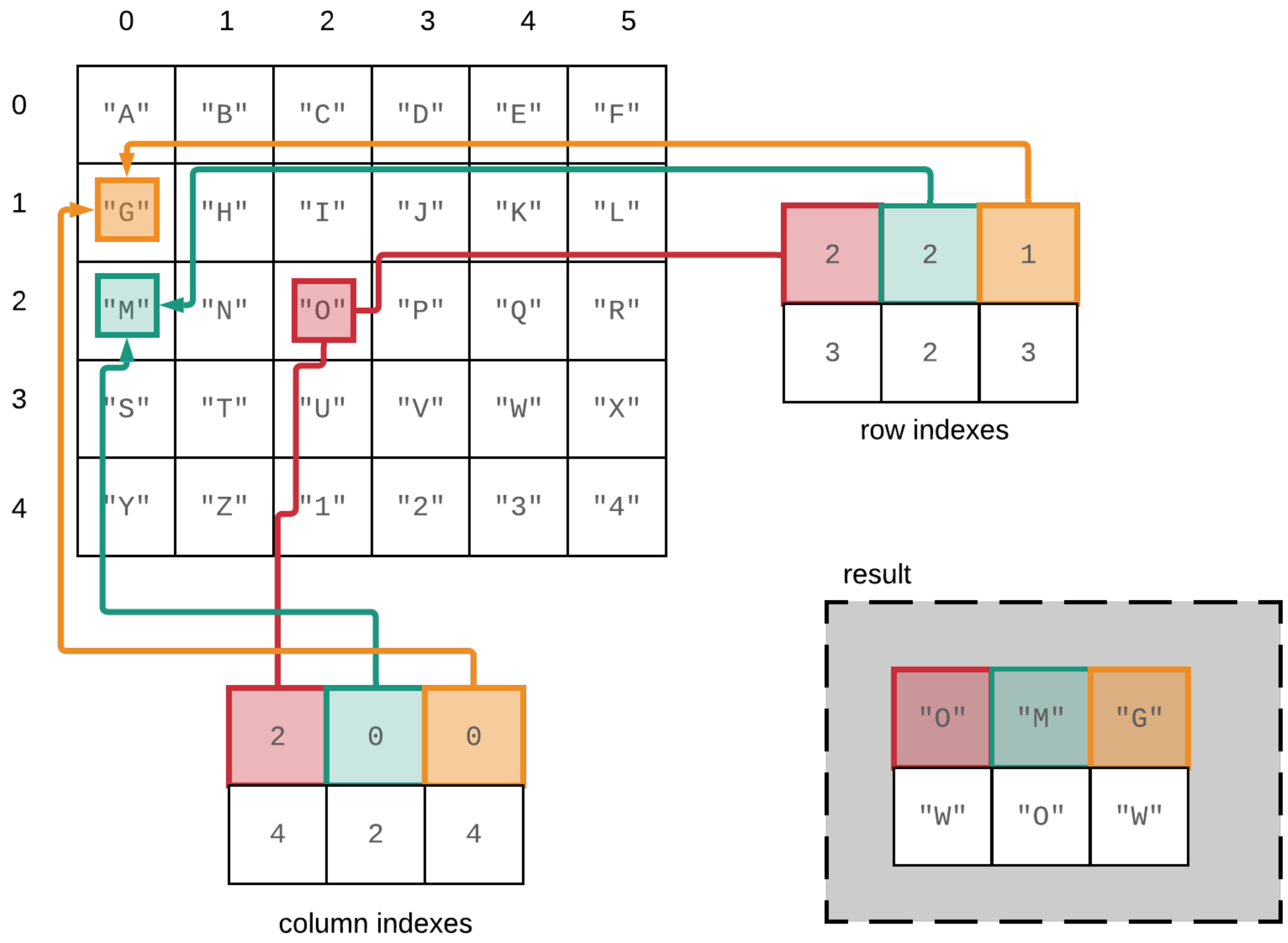
`exercises/numpy_broadcasting/broadcasting.ipynb`

- Broadcasting creates a view, and is an O(1) operation that does not require extra memory
- Rules of broadcasting
  - **1:** If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
  - **2:** If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
  - **3:** If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

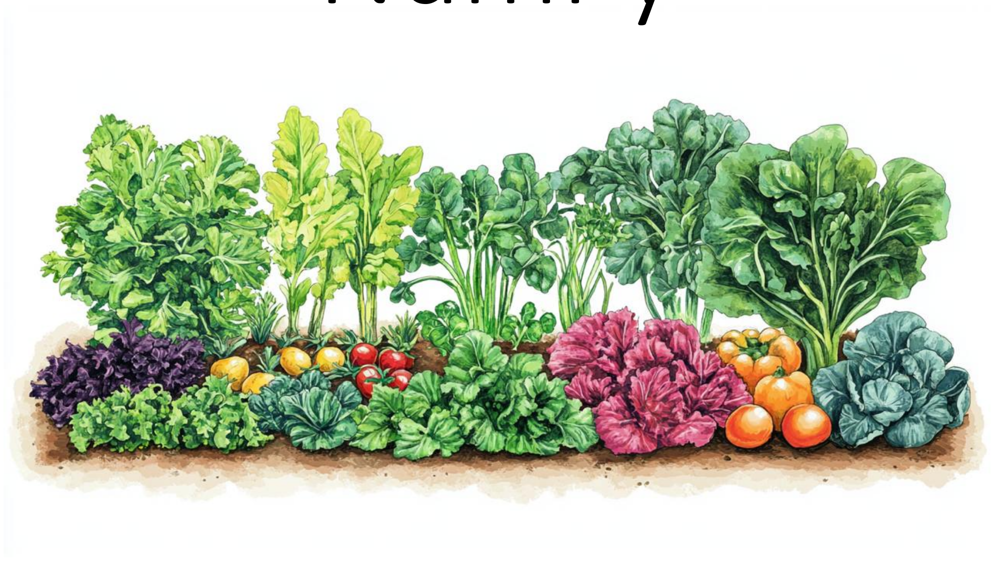
Up next: Tabular Data

Fancy indexing in NumPy – reference slide

A[[2, 2, 1], [2, 0, 0]]



NumPy

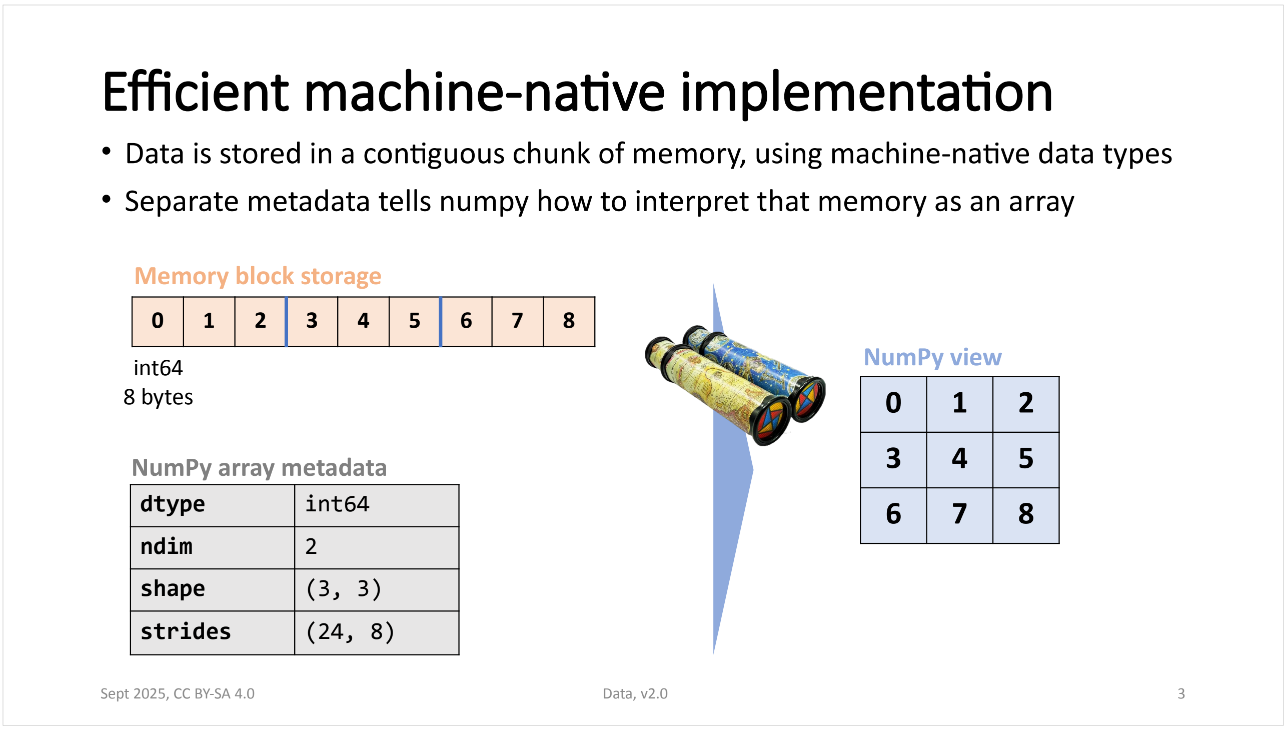


NumPy – huh, yeah – what's it good for?

- NumPy introduces a new data structure: **the array**

An array is a regular, N-dimensional grid of data of the same type, typically numerical data

- Great for storing homogeneous data, where every element in the array has the same meaning. E.g. images, sound, time series



Efficient machine-native implementation

- Data is stored in a contiguous chunk of memory, using machine-native data types
- Separate metadata tells numpy how to interpret that memory as an array

Memory block storage

0	1	2	3	4	5	6	7	8

int64
8 bytes

NumPy array metadata

`dtype`	`int64`
`ndim`	`2`
`shape`	`(3, 3)`
`strides`	`(24, 8)`

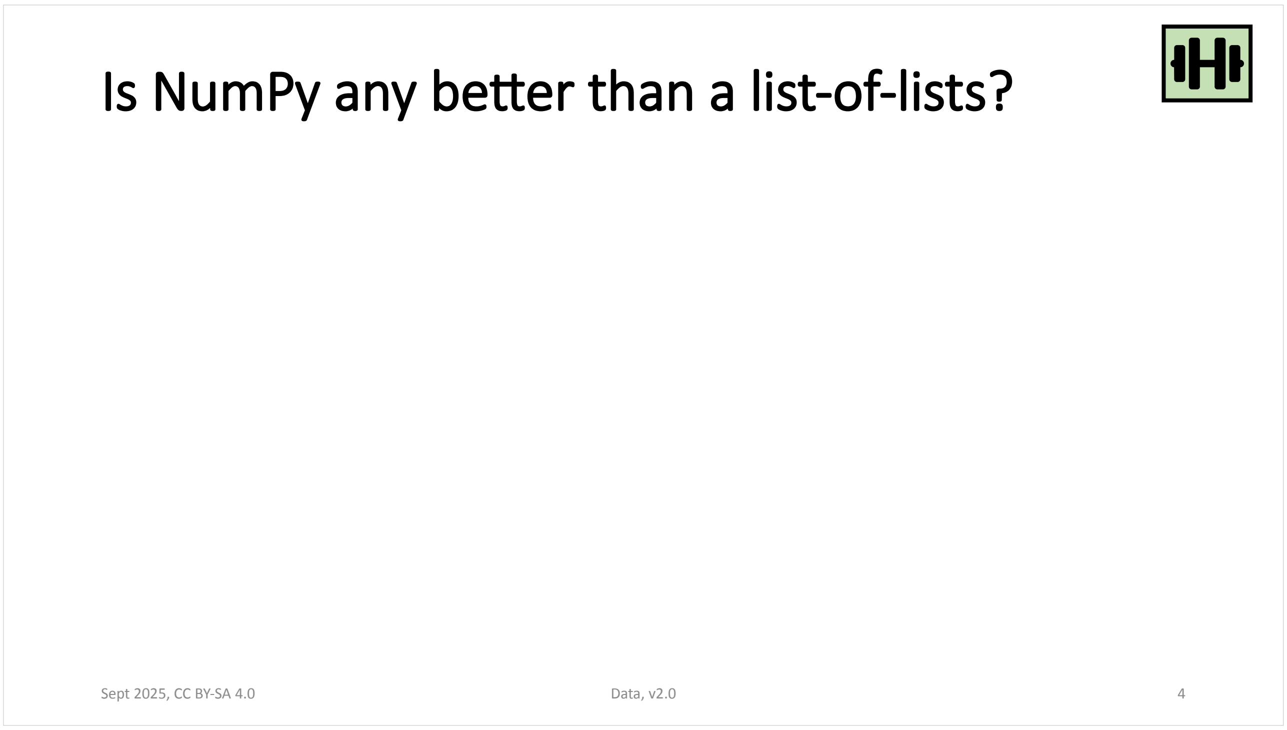
NumPy view

0	1	2
3	4	5
6	7	8

Sept 2025, CC BY-SA 4.0 Data, v2.0 3

Reminder of architecture class

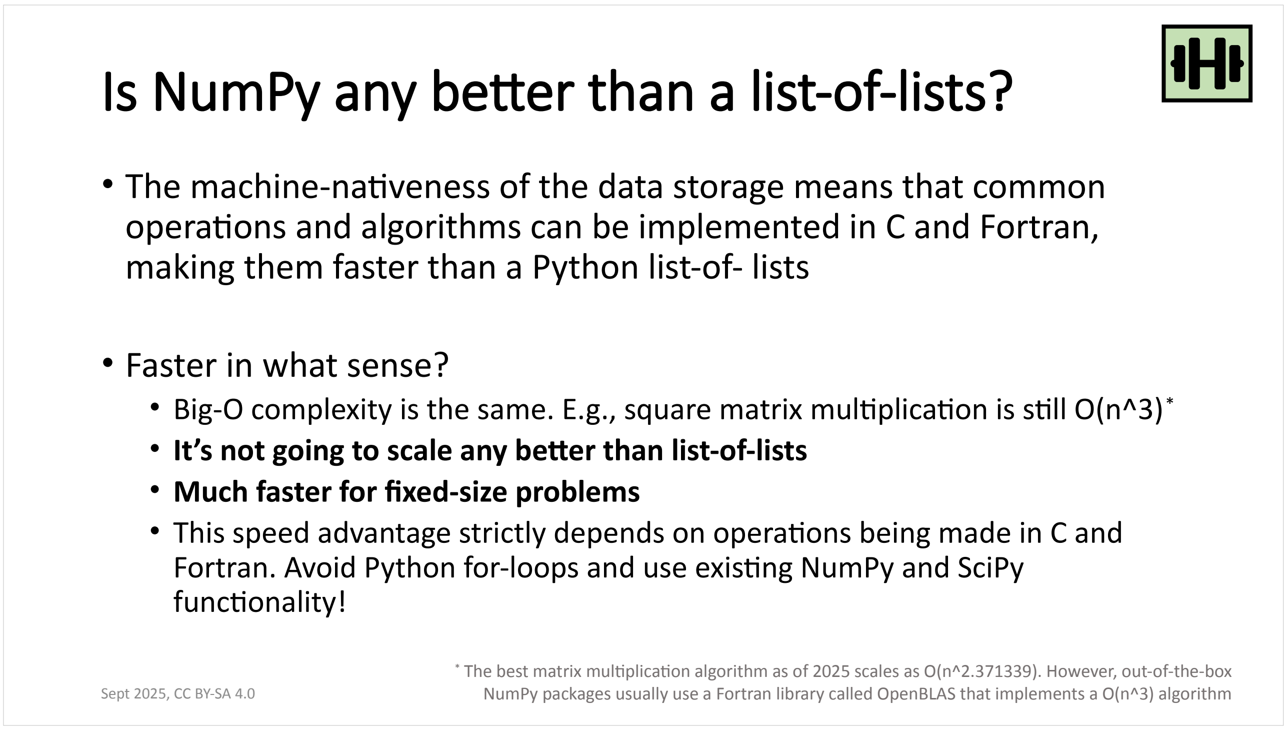
- The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but metadata as well
- a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value
- NumPy handles looping over array elements near-optimally—for example, taking strides into consideration to best utilize the computer's fast cache memory
- NumPy array - single pointer to one contiguous block of data
- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object



Is NumPy any better than a list-of-lists?

Sept 2025, CC BY-SA 4.0 Data, v2.0 4

- The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but metadata as well
- a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value
- NumPy handles looping over array elements near-optimally—for example, taking strides into consideration to best utilize the computer's fast cache memory
- NumPy array - single pointer to one contiguous block of data
- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object

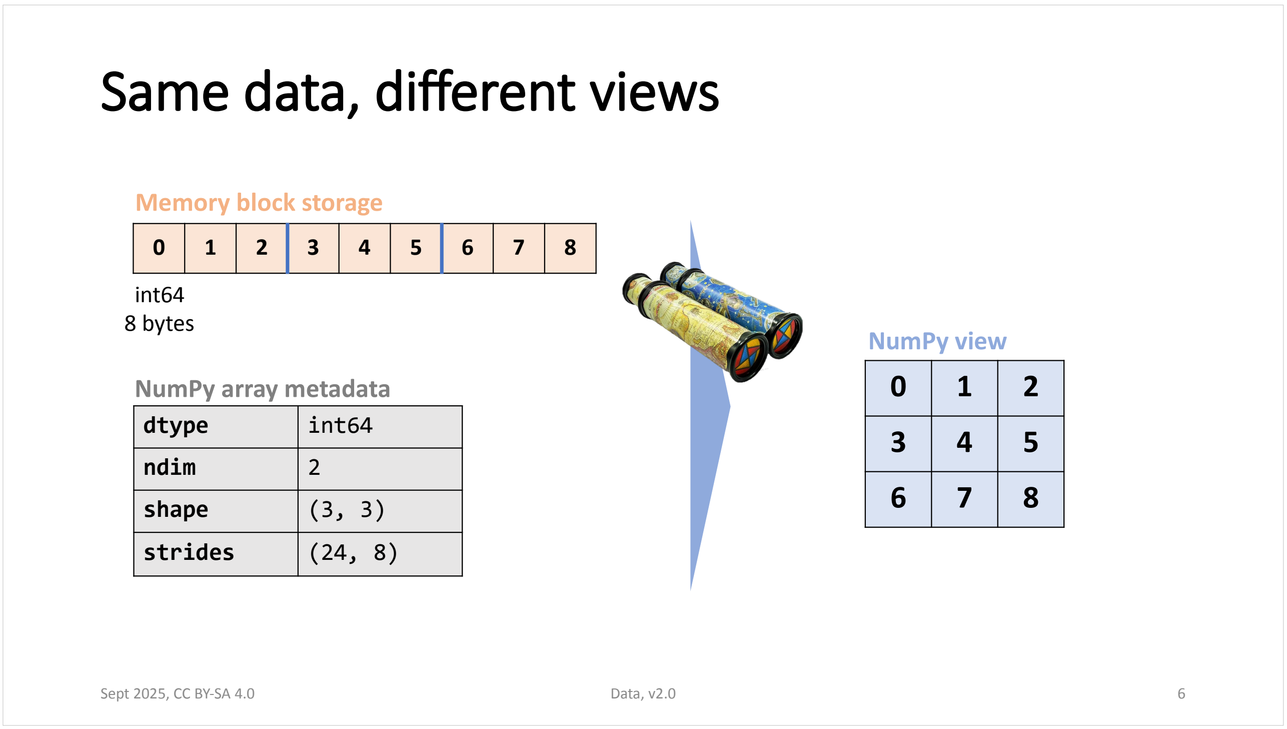


Is NumPy any better than a list-of-lists?

- The machine-nativeness of the data storage means that common operations and algorithms can be implemented in C and Fortran, making them faster than a Python list-of- lists
- Faster in what sense?
  - Big-O complexity is the same. E.g., square matrix multiplication is still O(n^3)*
  - **It's not going to scale any better than list-of-lists**
  - **Much faster for fixed-size problems**
  - This speed advantage strictly depends on operations being made in C and Fortran. Avoid Python for-loops and use existing NumPy and SciPy functionality!

* The best matrix multiplication algorithm as of 2025 scales as O(n^2.371339). However, out-of-the-box NumPy packages usually use a Fortran library called OpenBLAS that implements a O(n^3) algorithm

- The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but metadata as well
- a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value
- NumPy handles looping over array elements near-optimally—for example, taking strides into consideration to best utilize the computer's fast cache memory
- NumPy array - single pointer to one contiguous block of data
- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object



Same data, different views

Memory block storage

0	1	2	3	4	5	6	7	8

int64
8 bytes

NumPy array metadata

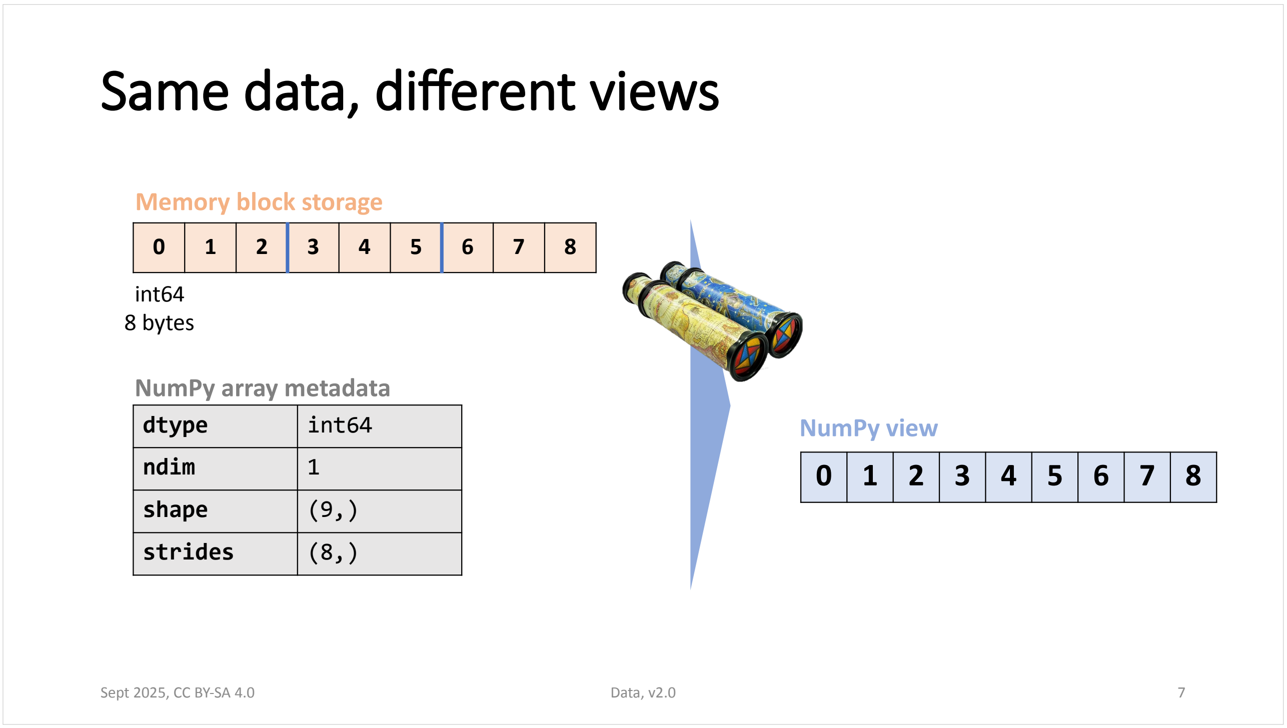
dtype	int64
ndim	2
shape	(3, 3)
strides	(24, 8)

NumPy view

0	1	2
3	4	5
6	7	8

Sept 2025, CC BY-SA 4.0 Data, v2.0 6

- The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but metadata as well
- a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value
- NumPy handles looping over array elements near-optimally—for example, taking strides into consideration to best utilize the computer's fast cache memory
- NumPy array - single pointer to one contiguous block of data
- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object



Same data, different views

Memory block storage

0	1	2	3	4	5	6	7	8

int64
8 bytes

NumPy array metadata

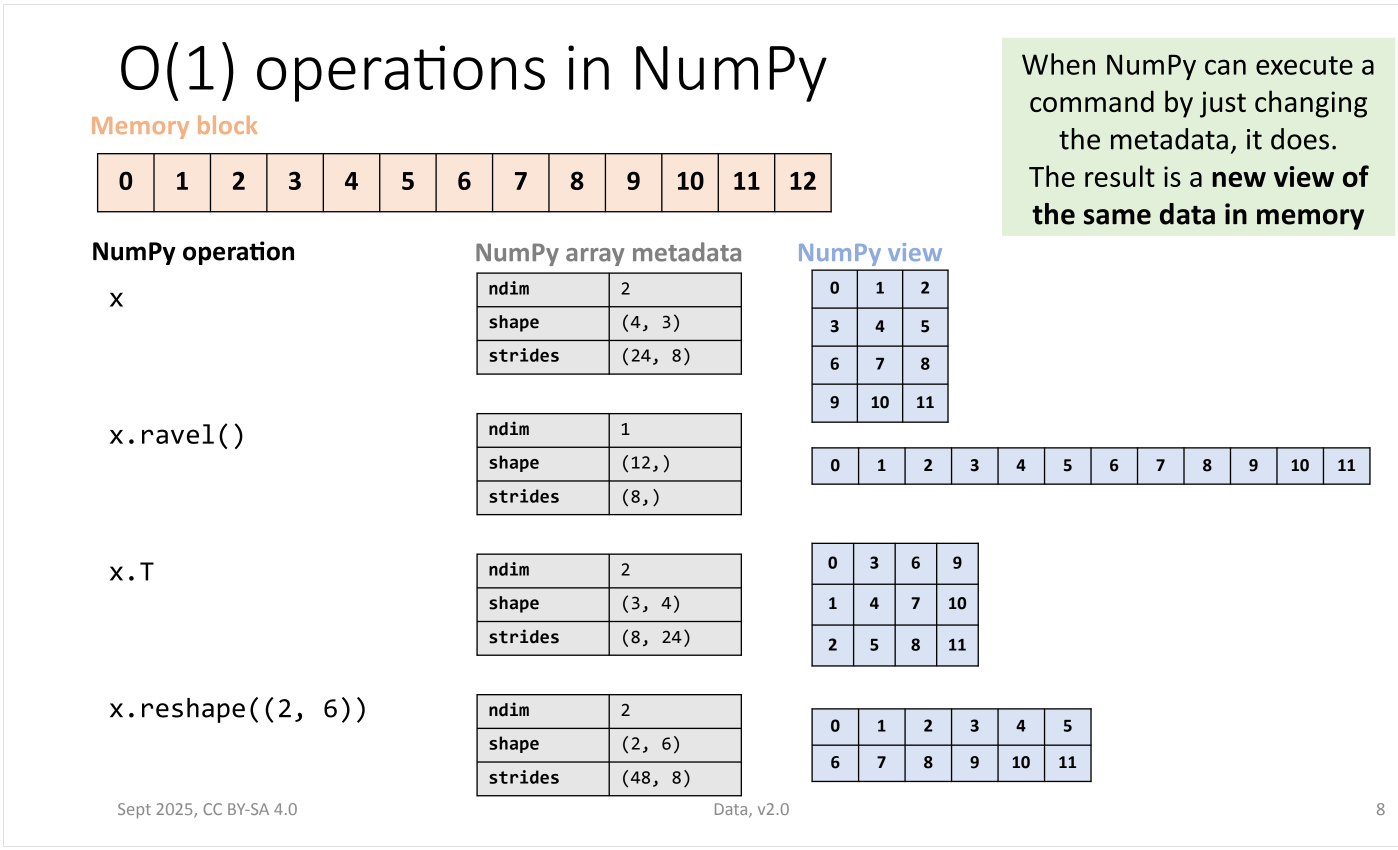
dtype	int64
ndim	1
shape	(9,)
strides	(8,)

NumPy view

0	1	2	3	4	5	6	7	8

Sept 2025, CC BY-SA 4.0 Data, v2.0 7

- The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but metadata as well
- a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value
- NumPy handles looping over array elements near-optimally—for example, taking strides into consideration to best utilize the computer’s fast cache memory
- NumPy array - single pointer to one contiguous block of data
- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object



O(1) operations in NumPy

When NumPy can execute a command by just changing the metadata, it does.
The result is a **new view of the same data in memory**

Memory block

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

NumPy operation — NumPy array metadata — NumPy view

```
x
```

ndim	2
shape	(4, 3)
strides	(24, 8)

0	**1**	**2**
3	**4**	**5**
6	**7**	**8**
9	**10**	**11**

```
x.ravel()
```

ndim	1
shape	(12,)
strides	(8,)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

```
x.T
```

ndim	2
shape	(3, 4)
strides	(8, 24)

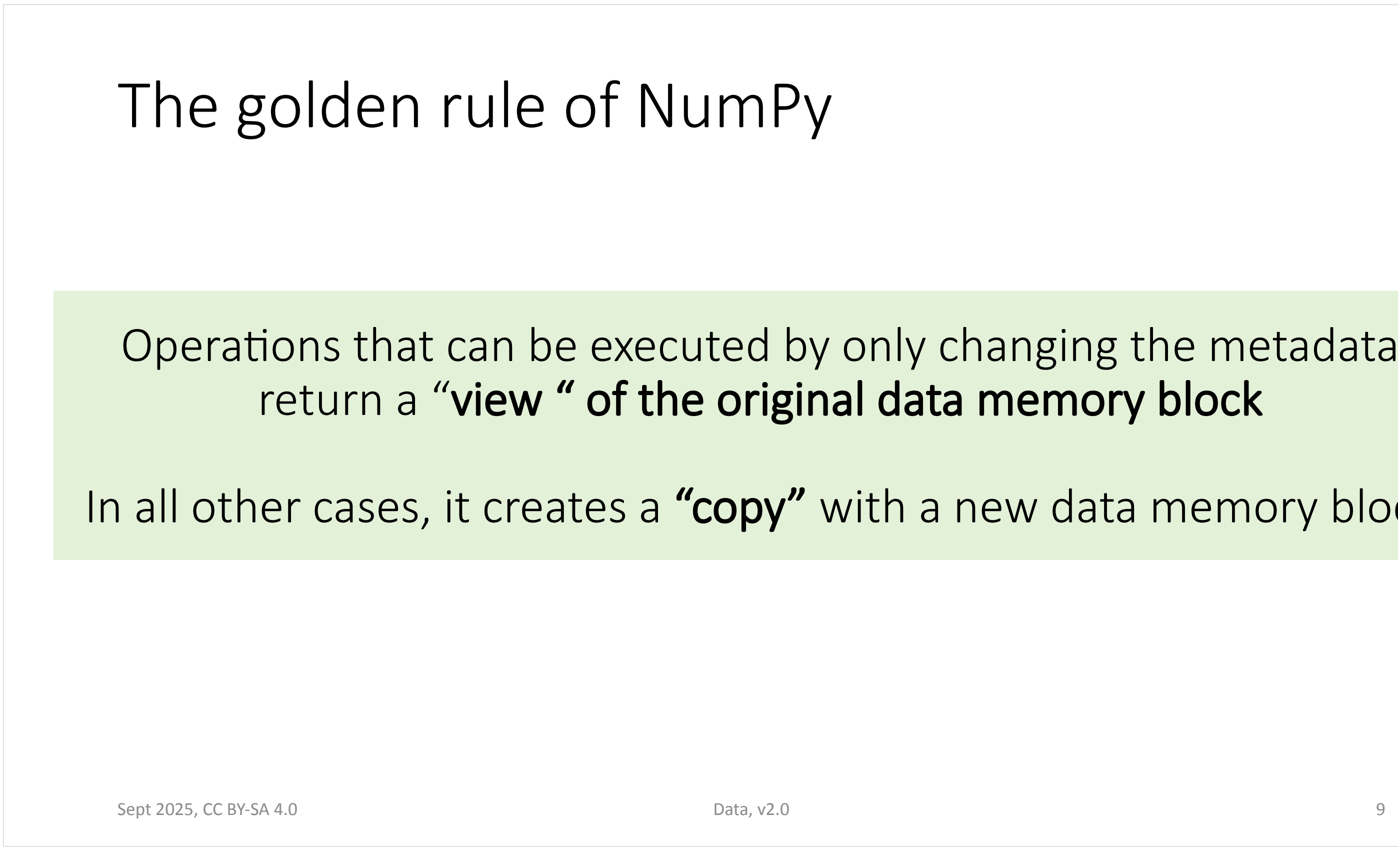
0	**3**	**6**	**9**
1	**4**	**7**	**10**
2	**5**	**8**	**11**

```
x.reshape((2, 6))
```

ndim	2
shape	(2, 6)
strides	(48, 8)

0	**1**	**2**	**3**	**4**	**5**
6	**7**	**8**	**9**	**10**	**11**

Point to make: there are a number of array operations that take no time at all!

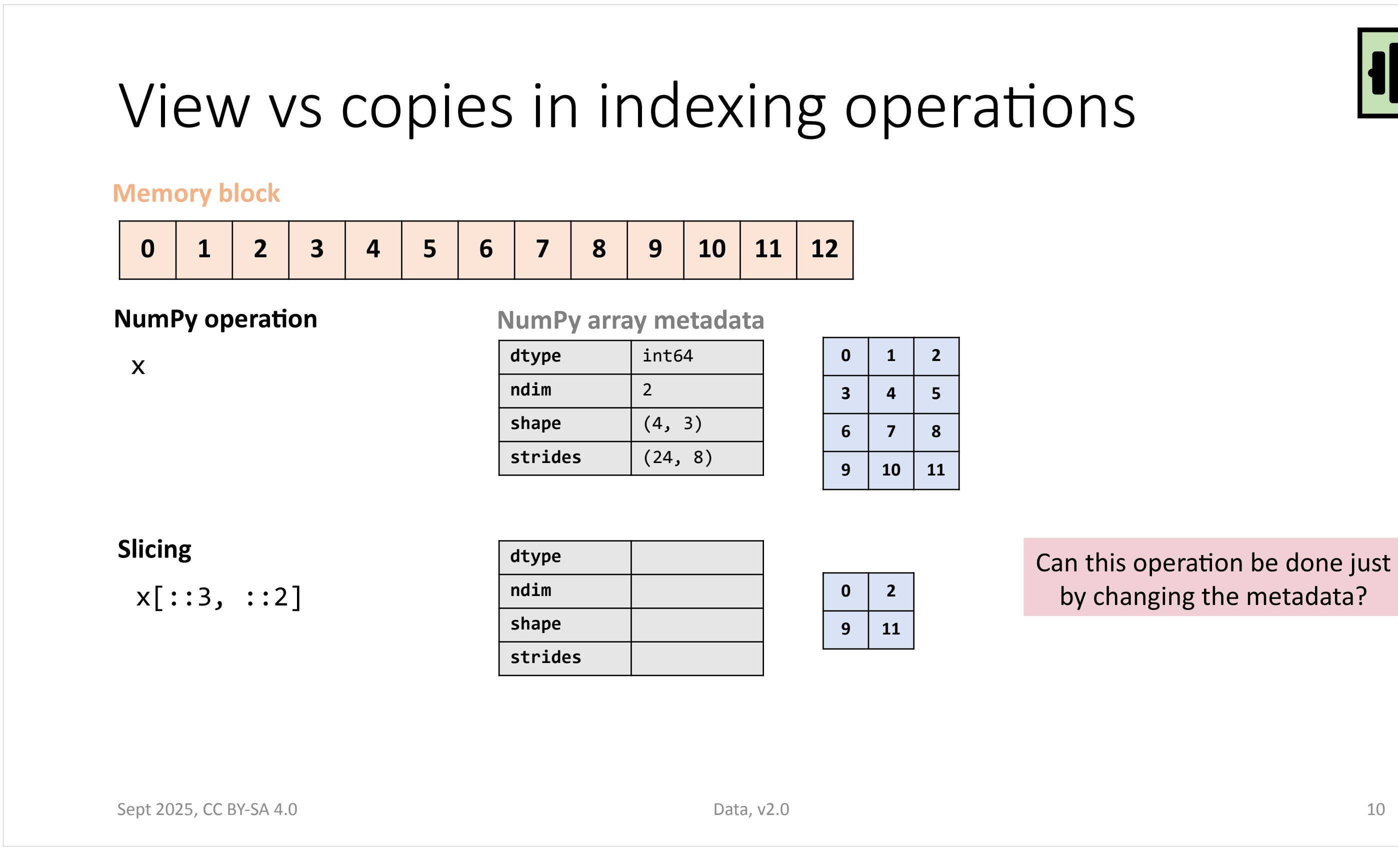


The golden rule of NumPy

Operations that can be executed by only changing the metadata
return a "**view " of the original data memory block**

In all other cases, it creates a **"copy"** with a new data memory bloc

You can always make a copy by explicitly calling .copy()



View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

```
x
```

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

0	1	2
3	4	5
6	7	8
9	10	11

Slicing

```
x[::3, ::2]
```

dtype	
ndim	
shape	
strides	

0	2
9	11

Can this operation be done just by changing the metadata?

How does the golden rule apply to indexing operations?

→ compare slices vs. fancy indexing

View vs copies in indexing operations

Memory block

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

NumPy operation

```
x
```

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

0	1	2
3	4	5
6	7	8
9	10	11

Slicing

```
x[::3, ::2]
```

dtype	int64
ndim	2
shape	(2, 2)
strides	(72, 16)

0	2
9	11

Can this operation be done just by changing the metadata? YES!

Can it be done for all slicing operations?

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

x

NumPy array metadata

Slicing always returns
a **view** of the original array

Slicing

`x[::3, ::2]`

dtype	int64
ndim	2
shape	(2, 2)
strides	(72, 16)

0	2
9	11

Can this operation be done just by changing the metadata?
YES!

Can it be done for all slicing operations?
YES!

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

x

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

0	1	2
3	4	5
6	7	8
9	10	11

Fancy indexing

$$\texttt{x[}\underbrace{\texttt{[0, 2, 3]}}_{\text{row indices}}\texttt{, }\underbrace{\texttt{[1, 2, 1]}}_{\text{column indices}}\texttt{]}$$

dtype	
ndim	
shape	
strides	

1	10	8

Can this operation be done just by changing the metadata?

View vs copies in indexing operations

Memory block

0	1	2	3	4	5	6	7	8	9	10	11	12

NumPy operation

x

NumPy array metadata

Fancy indexing always returns a **copy** of the original array

Fancy indexing

$$\underbrace{\texttt{x[[0, 2, 3],}}_{\text{row indices}}\ \underbrace{\texttt{[1, 2, 1]]}}_{\text{column indices}}$$

dtype	int64
ndim	1
shape	(3,)
strides	???

1	10	8

Can this operation be done just by changing the metadata?
NO!

View or copy? Quiz

Exercise

`exercises/numpy_view_or_copy/`
`view_or_copy_interactive.ipynb`

Changing one view changes them all

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

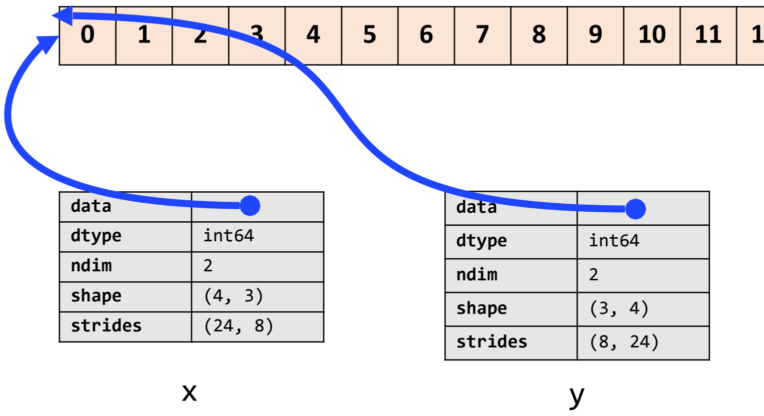
```
x
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
y = x.T
```

```
y
```

```
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```

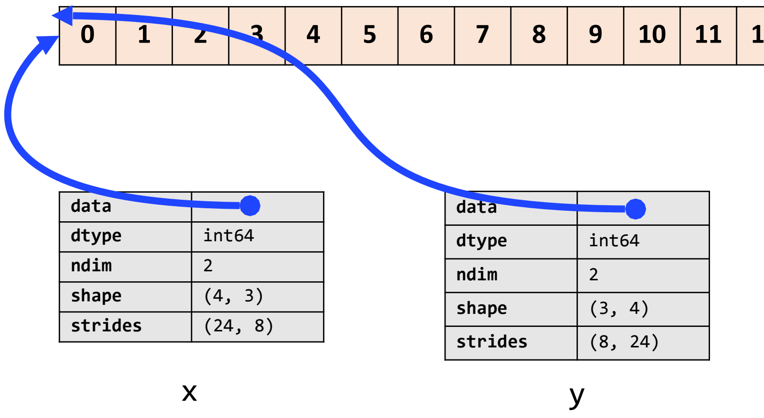


data	
dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

data	
dtype	int64
ndim	2
shape	(3, 4)
strides	(8, 24)

Changing one view changes them all

```
y[0, 0] = 13
```



data	
dtype	int64
ndim	2
shape	(4, 3)
strides	(24, 8)

x

data	
dtype	int64
ndim	2
shape	(3, 4)
strides	(8, 24)

y

Changing one view changes them all

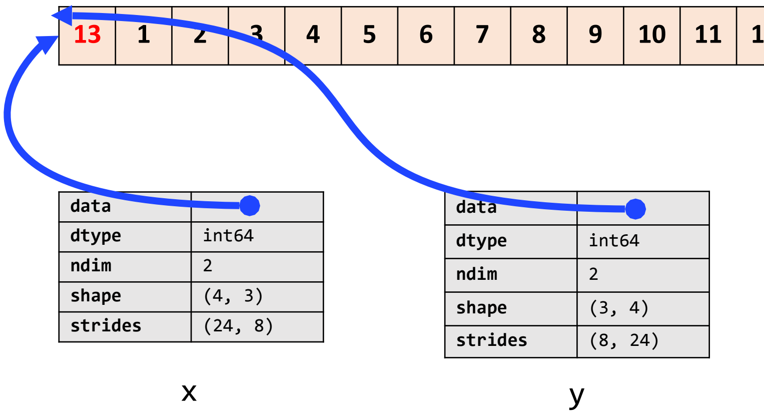
```
y[0, 0] = 13
```

```
y
```

```
array([[13,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```

```
x
```

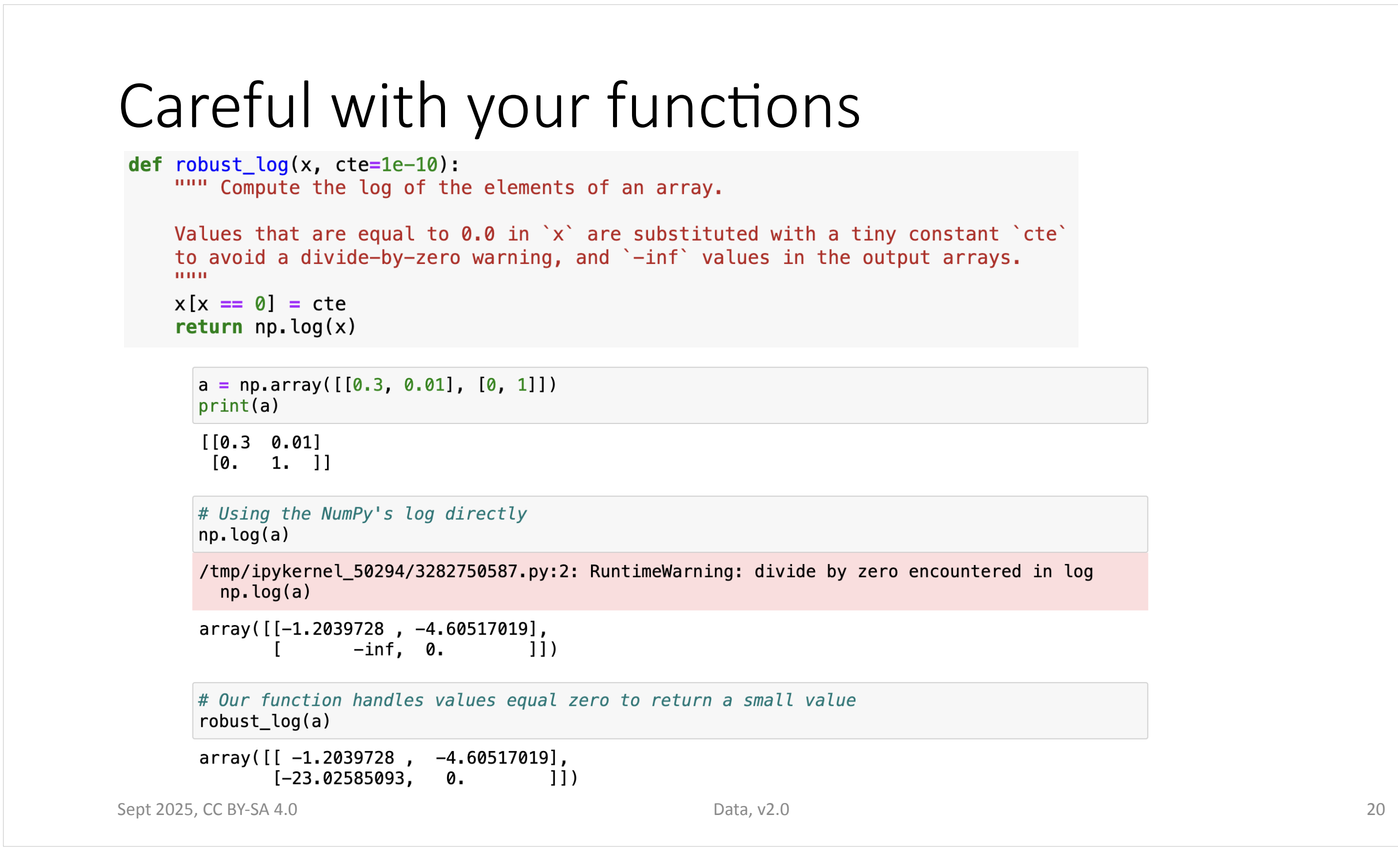
```
array([[13,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```



Careful with your functions

```python
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x[x == 0] = cte
    return np.log(x)
```



Careful with your functions

```python
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x[x == 0] = cte
    return np.log(x)
```

```python
a = np.array([[0.3, 0.01], [0, 1]])
print(a)
```

```
[[0.3  0.01]
 [0.   1.  ]]
```

```python
# Using the NumPy's log directly
np.log(a)
```

```
/tmp/ipykernel_50294/3282750587.py:2: RuntimeWarning: divide by zero encountered in log
  np.log(a)
```

```
array([[-1.2039728 , -4.60517019],
       [       -inf,  0.        ]])
```

```python
# Our function handles values equal zero to return a small value
robust_log(a)
```

```
array([[ -1.2039728 ,  -4.60517019],
       [-23.02585093,   0.        ]])
```

Example .copy() usage

What can go wrong with this function?

Careful with your functions

```
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x[x == 0] = cte
    return np.log(x)
```

```
a = np.array([[0.3, 0.01], [0, 1]])
b = a[1, :]  # A view of `a`
print(b)
```

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

```
robust_log(b)
```

```
array([-23.02585093,   0.        ])
```

```
b
```

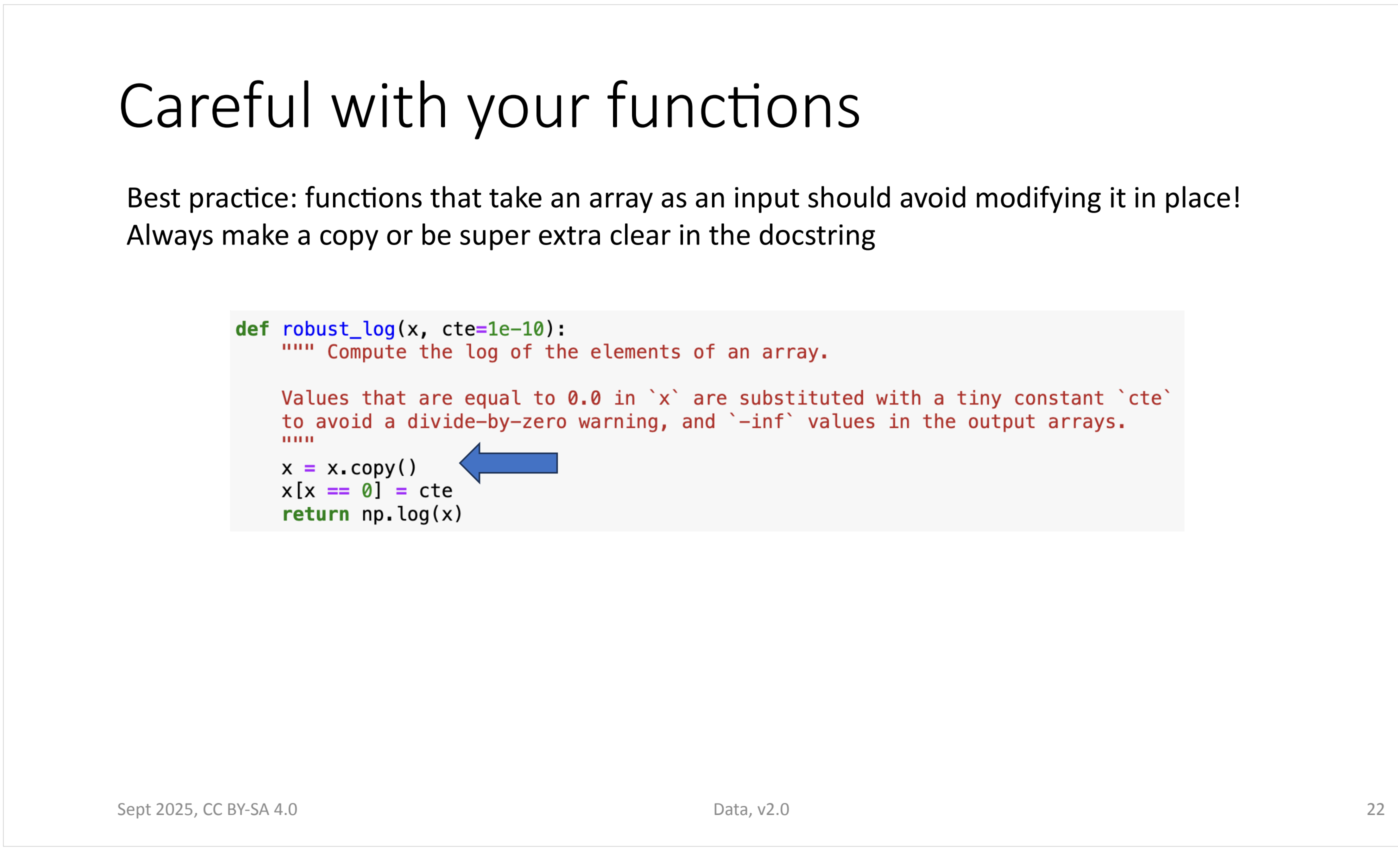
```
array([1.e-10, 1.e+00])
```

The input array has been modifie

```
a
```

```
array([[3.e-01, 1.e-02],
       [1.e-10, 1.e+00]])
```

… and so have all other views of the same data!



Careful with your functions

Best practice: functions that take an array as an input should avoid modifying it in place!
Always make a copy or be super extra clear in the docstring

```python
def robust_log(x, cte=1e-10):
    """ Compute the log of the elements of an array.

    Values that are equal to 0.0 in `x` are substituted with a tiny constant `cte`
    to avoid a divide-by-zero warning, and `-inf` values in the output arrays.
    """
    x = x.copy()
    x[x == 0] = cte
    return np.log(x)
```

Discuss pros and cons with them

Pros: security that data will be always a copy.

Cons: redundancy in memory, problem for big arrays

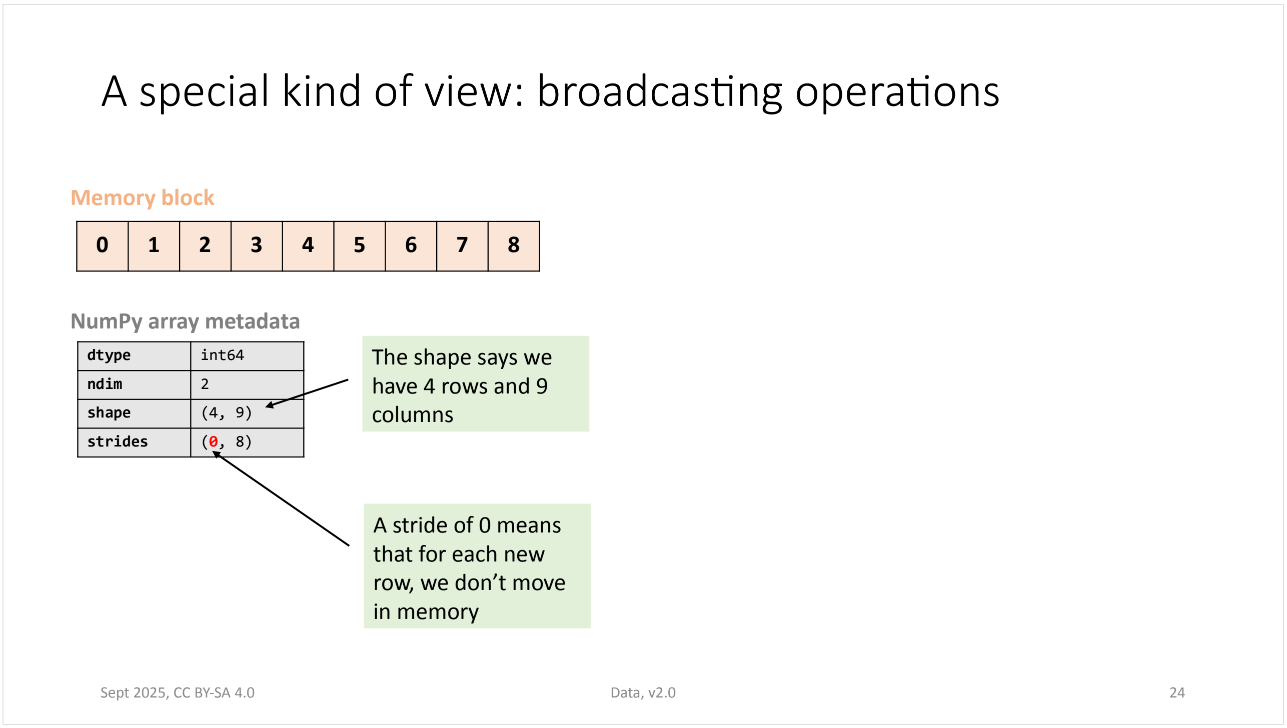
NumPy views and copies summary

View

- There can be multiple views of the same memory block, interpreted as different arrays
- Slicing returns a view
- In-place operations on a view modify the memory block and all of its views

Copy

- When a copy of an array needs to be created, it allocates a separate memory block and associates it with new metadata
- Fancy indexing always returns copies
- A copy can be forced with .copy()



A special kind of view: broadcasting operations

Memory block

0	1	2	3	4	5	6	7	8

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 9)
strides	(0, 8)

The shape says we have 4 rows and 9 columns

A stride of 0 means that for each new row, we don't move in memory

Sept 2025, CC BY-SA 4.0 Data, v2.0 24

Special case, when one of the strides is 0

✉ duplicates without occupying more memory

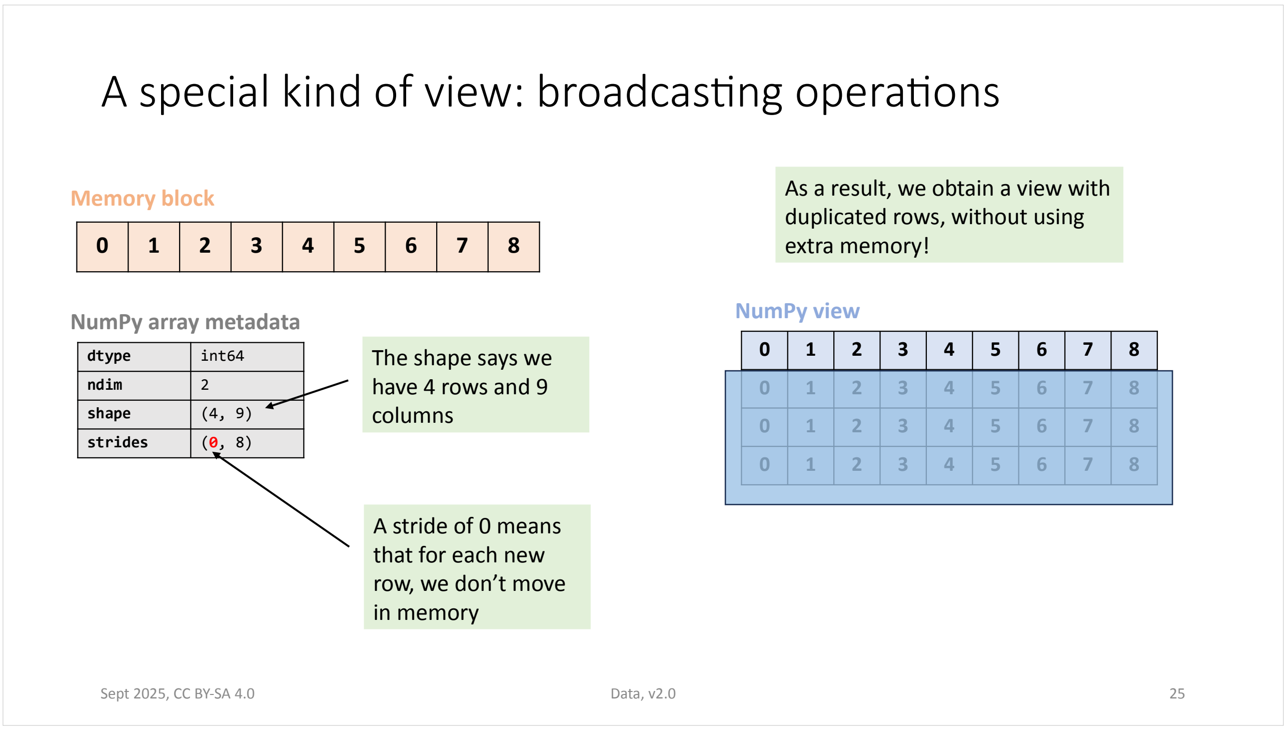
Transition, other operations that can

Braodcasting

Imagine what happens if we have a stride of 0

It allows us to replicate the sae row without allocating extra memory

What happens when it’s a 0?



A special kind of view: broadcasting operations

Memory block

0	1	2	3	4	5	6	7	8

NumPy array metadata

dtype	int64
ndim	2
shape	(4, 9)
strides	(0, 8)

The shape says we have 4 rows and 9 columns

A stride of 0 means that for each new row, we don't move in memory

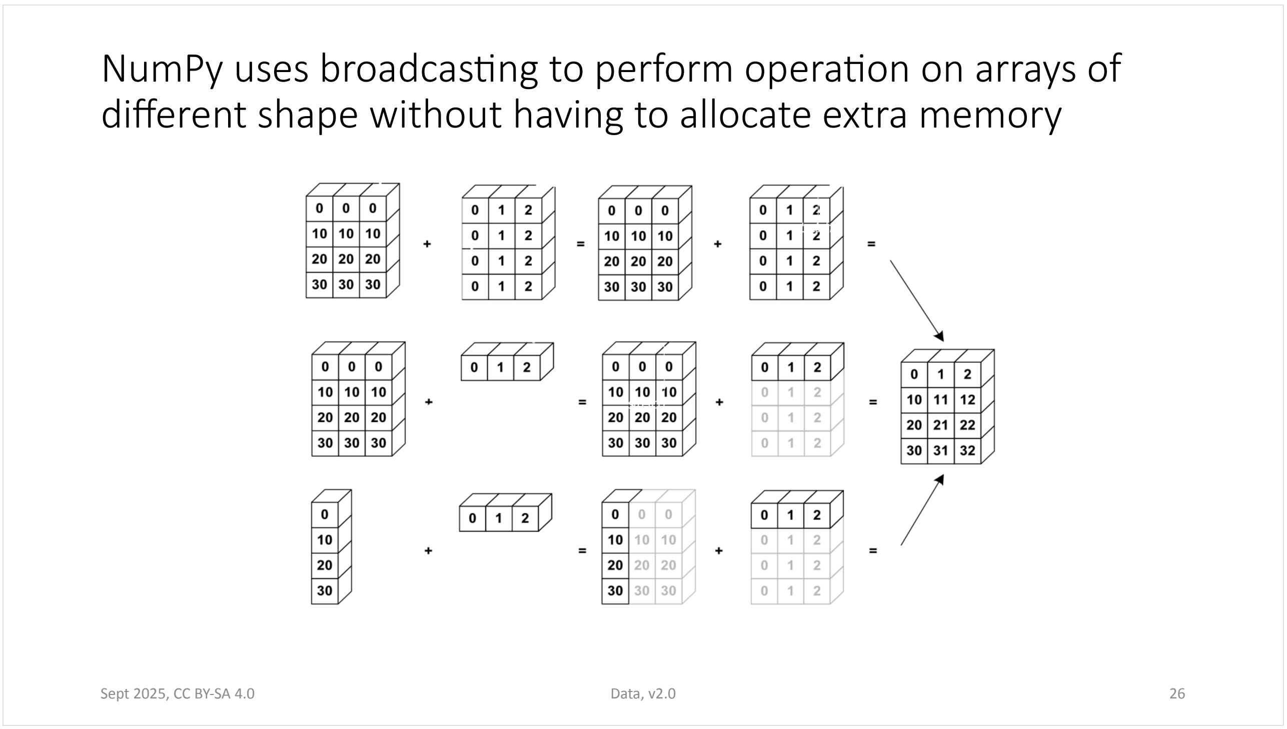
As a result, we obtain a view with duplicated rows, without using extra memory!

NumPy view

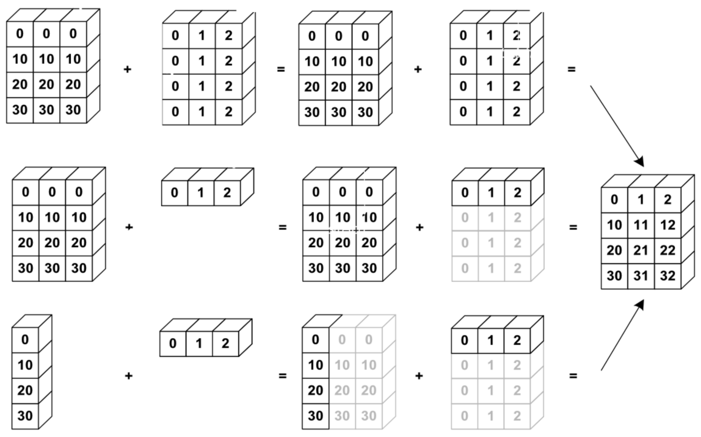
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

Sept 2025, CC BY-SA 4.0 Data, v2.0 25

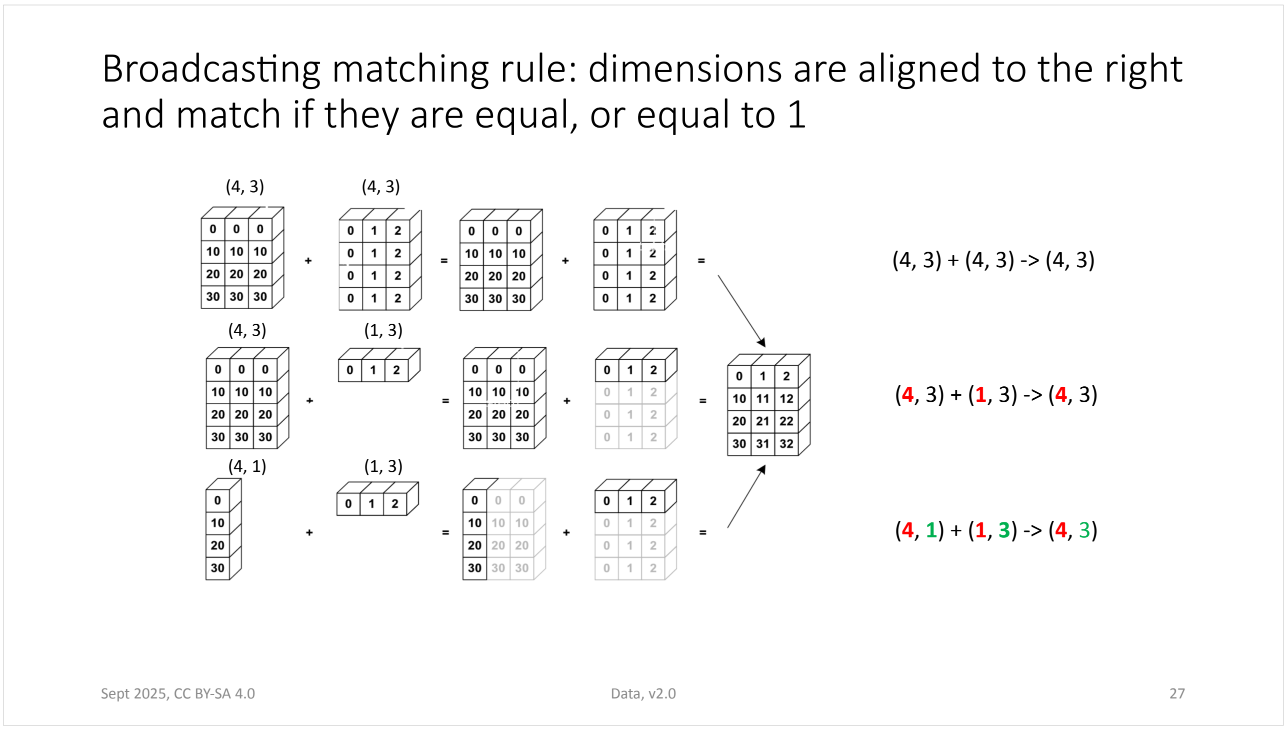
Stride == 0 ✉ then we are not actually moving in memory, same memory, we are looping over the same thing



NumPy uses broadcasting to perform operation on arrays of different shape without having to allocate extra memory



Again the kaleidoscope metaphor, it’s an aritifical way to replicate something



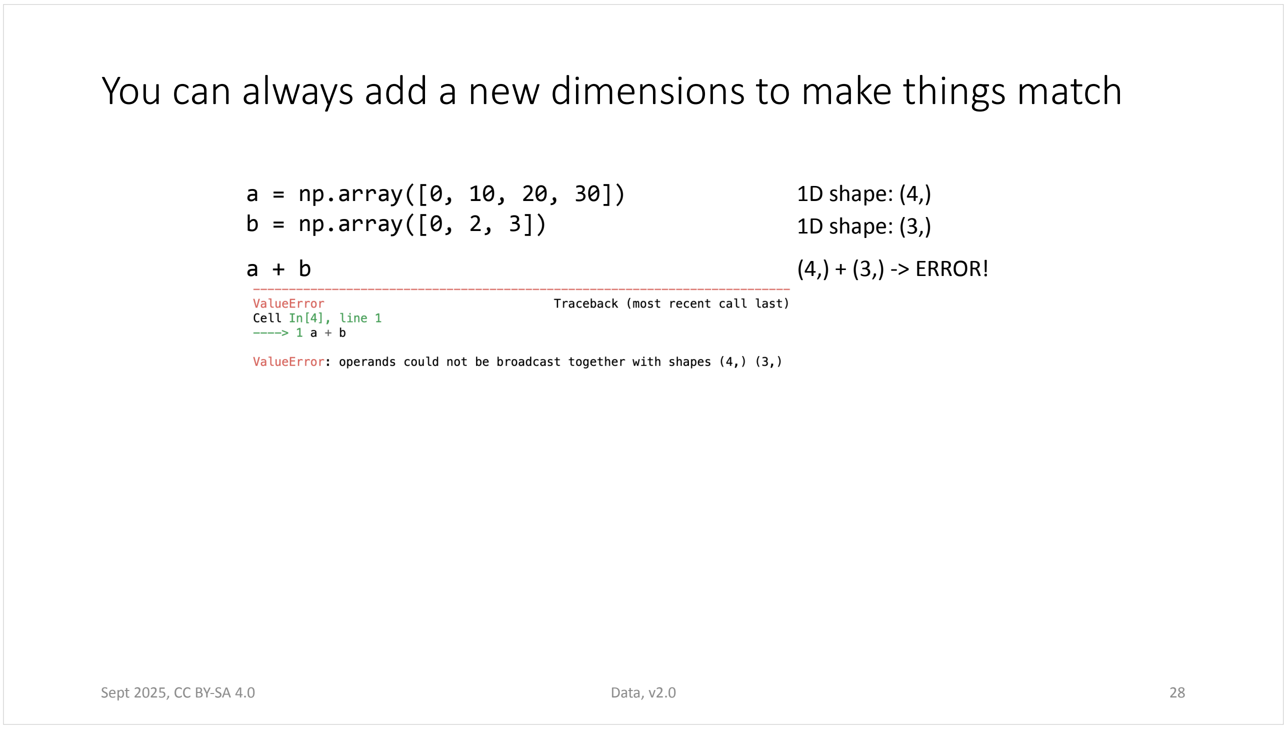
Broadcasting matching rule: dimensions are aligned to the right and match if they are equal, or equal to 1

(4, 3) + (4, 3) -> (4, 3)

(4, 3) + (1, 3) -> (4, 3)

(4, 1) + (1, 3) -> (4, 3)

Again the kaleidoscope metaphor, it’s an aritifical way to replicate something



You can always add a new dimensions to make things match

```
a = np.array([0, 10, 20, 30])
b = np.array([0, 2, 3])

a + b
```

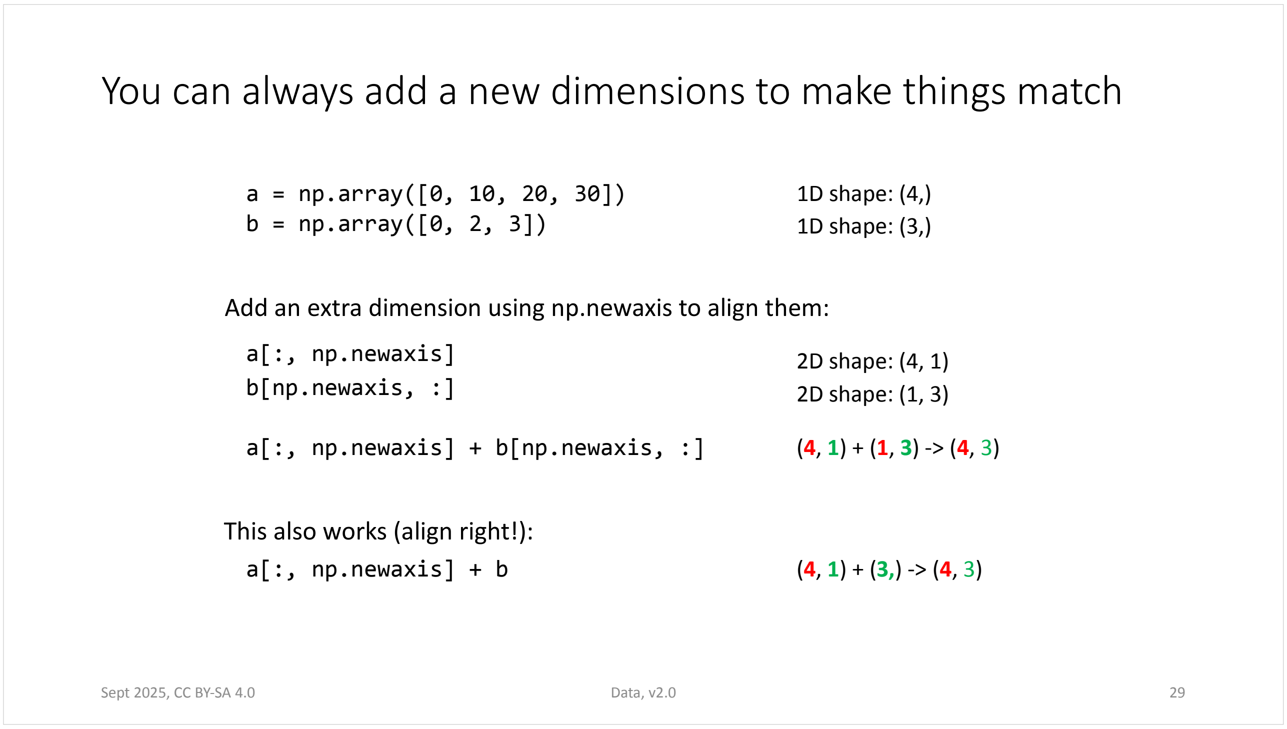
1D shape: (4,)
1D shape: (3,)
(4,) + (3,) -> ERROR!

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 a + b

ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

Sept 2025, CC BY-SA 4.0 Data, v2.0 28

Again the kaleidoscope metaphor, it’s an aritifical way to replicate something



You can always add a new dimensions to make things match

```
a = np.array([0, 10, 20, 30])
b = np.array([0, 2, 3])
```

1D shape: (4,)
1D shape: (3,)

Add an extra dimension using np.newaxis to align them:

```
a[:, np.newaxis]
b[np.newaxis, :]
```

2D shape: (4, 1)
2D shape: (1, 3)

```
a[:, np.newaxis] + b[np.newaxis, :]
```

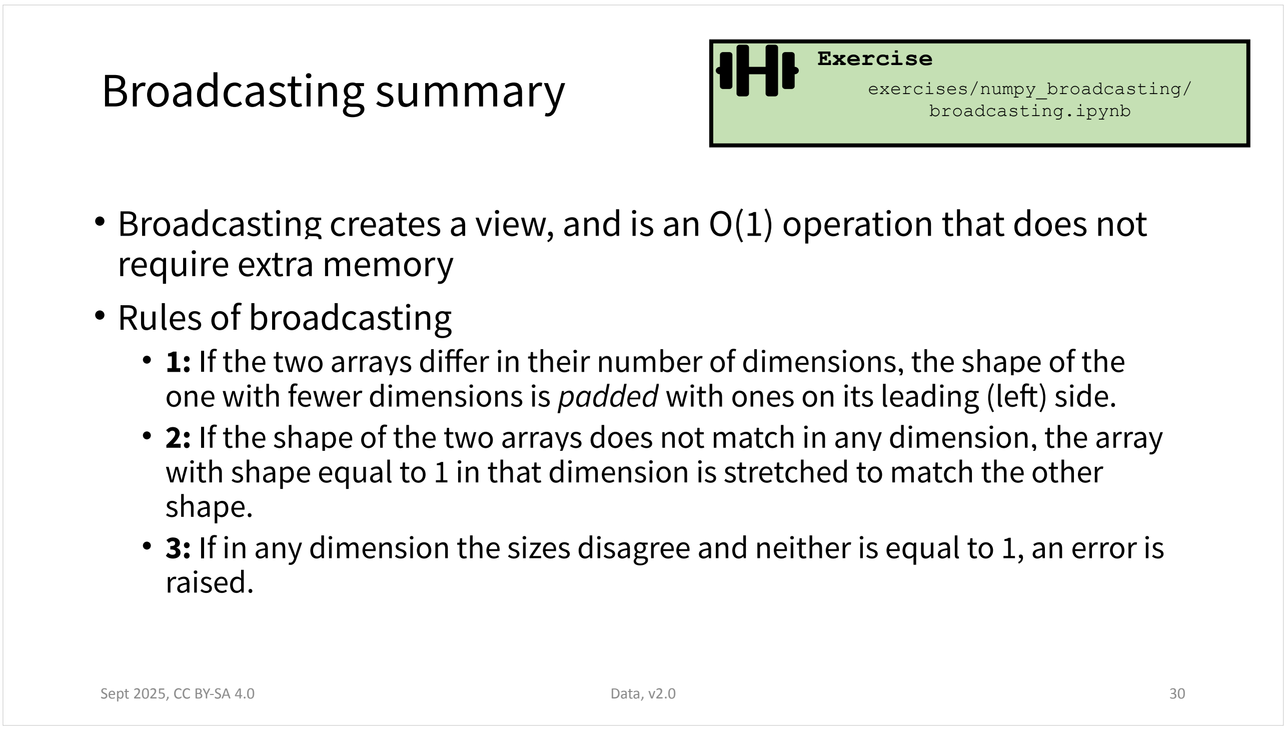
(**4**, **1**) + (**1**, **3**) -> (**4**, 3)

This also works (align right!):

```
a[:, np.newaxis] + b
```

(**4**, **1**) + (**3,**) -> (**4**, 3)

Again the kaleidoscope metaphor, it’s an aritifical way to replicate something



Broadcasting summary

Exercise
exercises/numpy_broadcasting/
broadcasting.ipynb

- Broadcasting creates a view, and is an O(1) operation that does not require extra memory
- Rules of broadcasting
  - **1:** If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
  - **2:** If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
  - **3:** If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

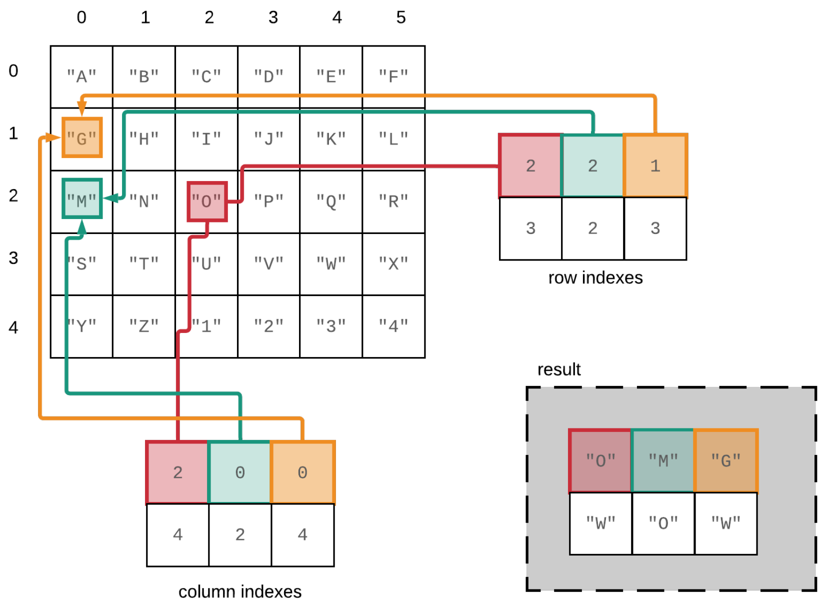
Sept 2025, CC BY-SA 4.0 Data, v2.0 30

Show 1,2 examples and the notebook with the rules/ their implementation

Up next: Tabular Data

Fancy indexing in NumPy – reference slide

A[[2, 2, 1], [2, 0, 0]]



0 1 2 3 4 5
0 "A" "B" "C" "D" "E" "F"
1 "G" "H" "I" "J" "K" "L"
2 "M" "N" "O" "P" "Q" "R"
3 "S" "T" "U" "V" "W" "X"
4 "Y" "Z" "1" "2" "3" "4"
2 2 1
3 2 3
row indexes
result
"O" "M" "G"
"W" "O" "W"
2 0 0
4 2 4
column indexes