Week 4 – Profiling and High-Performance NumPy

02613 Python and High-Performance Computing

Last week on 02613

- Any questions about caches and memory hierarchy?

Today

1. NumPy

2. Profiling

NumPy

Quantum Computing

- QuTiP
- PyQuil
- Qiskit
- PennyLane

Statistical Computing

- Pandas
- statsmodels
- Xarray
- Seaborn

Signal Processing

- SciPy
- PyWavelets
- python-control

Image Processing

- Scikit-image
- OpenCV
- Mahotas

Graphs and Networks

- NetworkX
- graph-tool
- igraph
- PyGSP

Astronomy

- AstroPy
- SunPy
- SpacePy

Cognitive Psychology

- PsychoPy

Bioinformatics

- BioPython
- Scikit-Bio
- PyEnsembl
- ETE

Bayesian Inference

- PyStan
- PyMC3
- ArviZ
- emcee

Mathematical Analysis

- SciPy
- SymPy
- cvxpy
- FEniCS

Chemistry

- Cantera
- MDAnalysis
- RDKit
- PyBaMM

Geoscience

- Pangeo
- Simpeg
- ObsPy
- Fatiando a Terra

Geographic Processing

- Shapely
- GeoPandas
- Folium

Architecture & Engineering

- COMPAS
- City Energy Analyst
- Sverchok

numpy.org

PyTorch

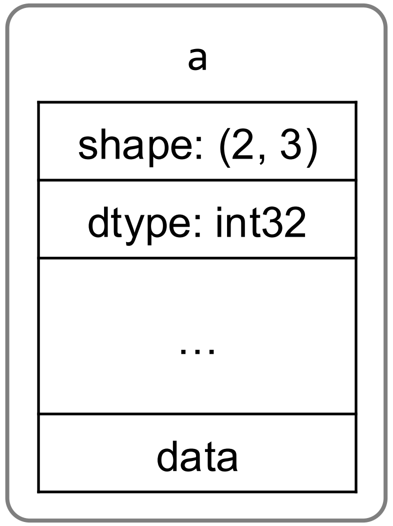
What is a NumPy array

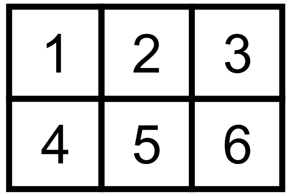
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
```

1	2	3
4	5	6

What is a NumPy array

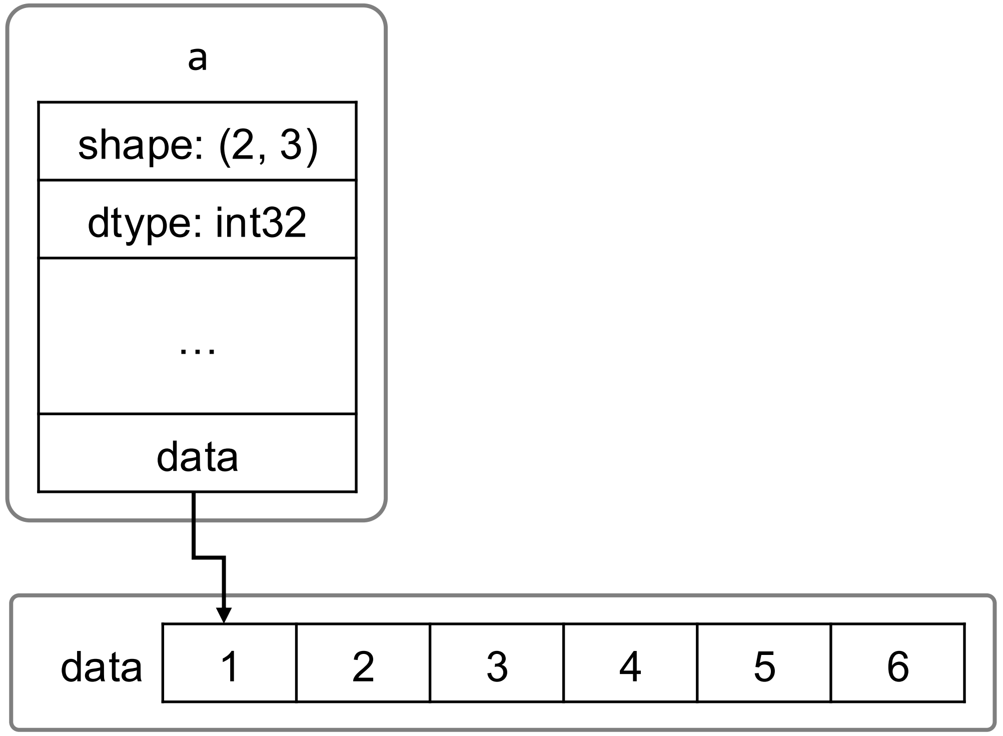
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
```

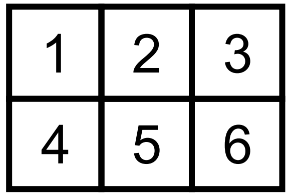


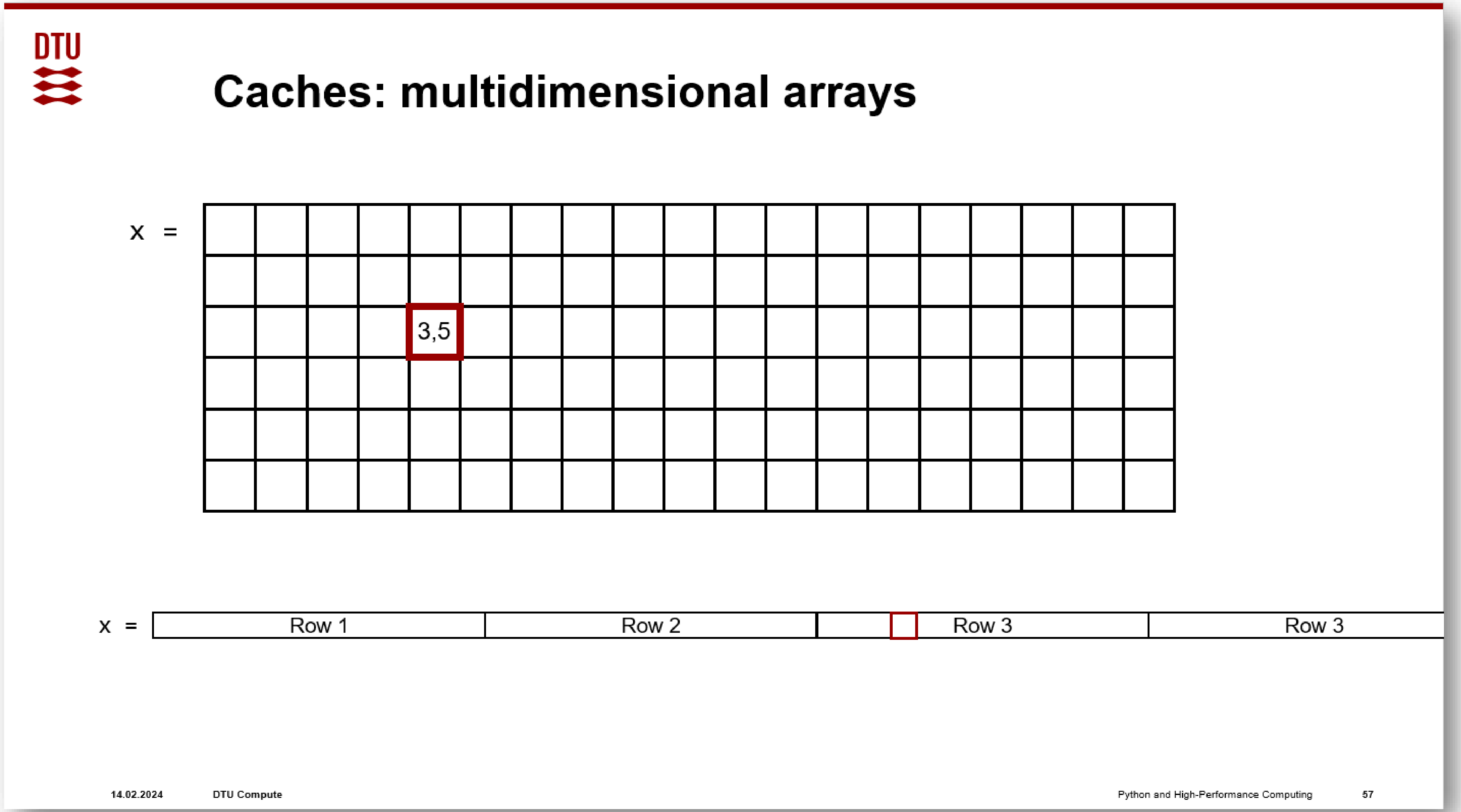


What is a NumPy array

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



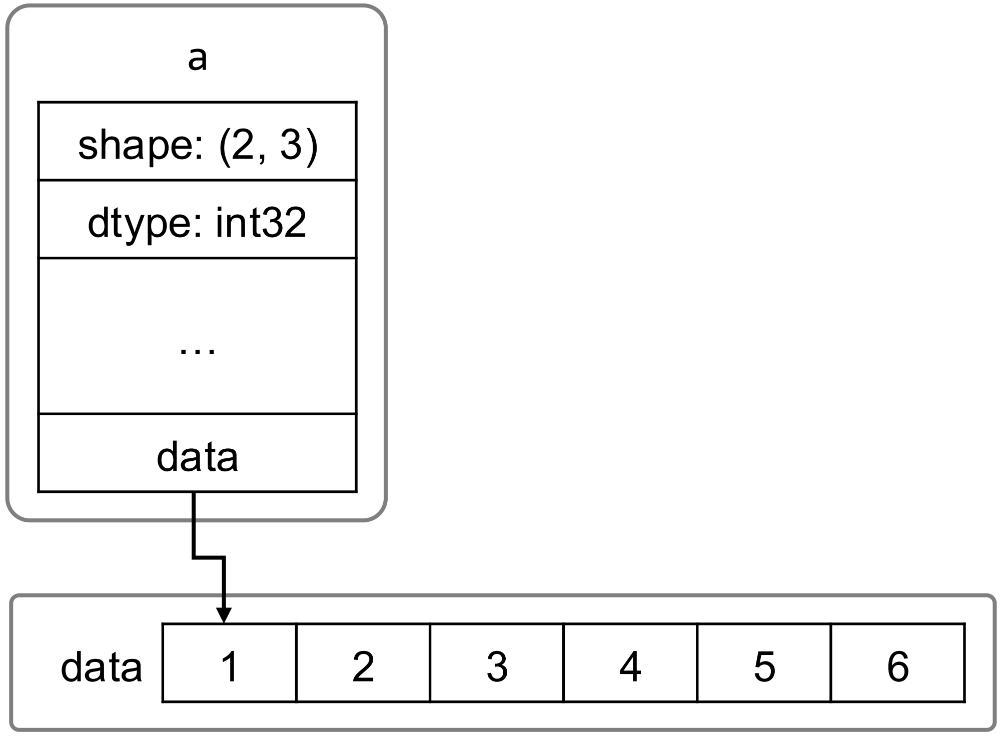


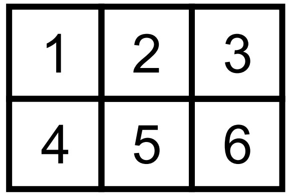


Caches: multidimensional arrays

What is a NumPy array

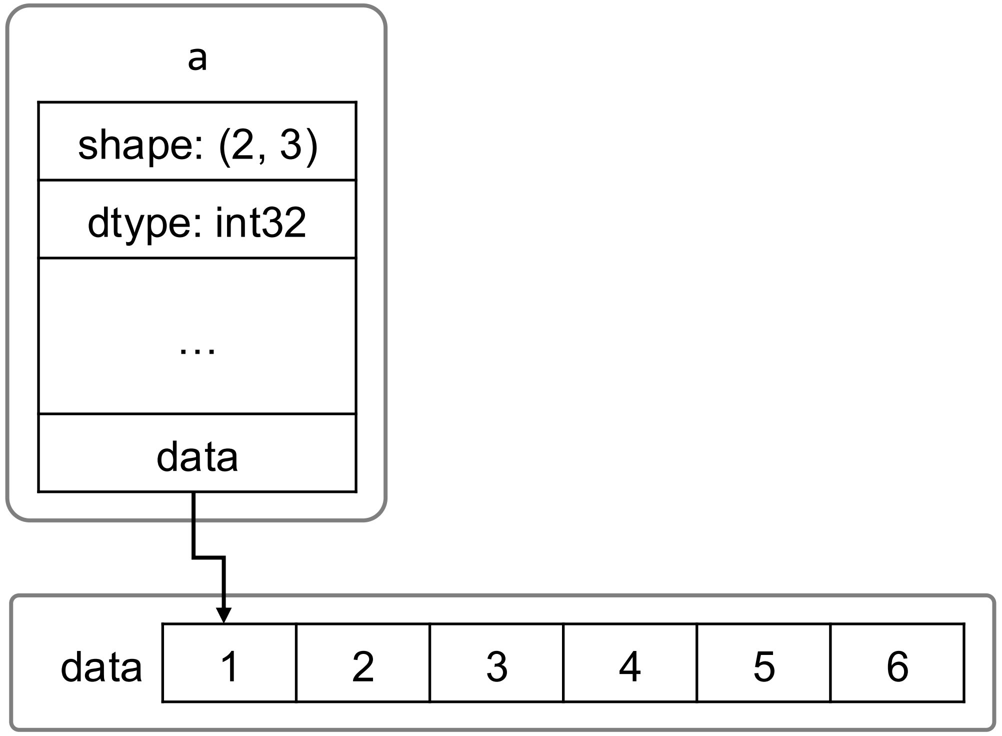
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
```

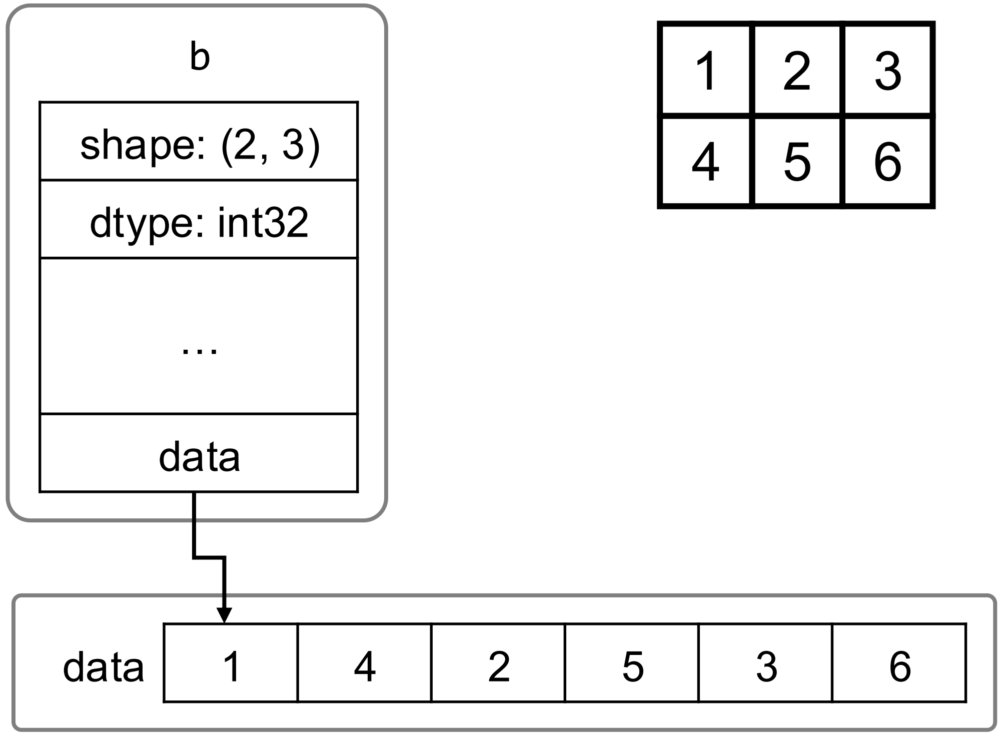


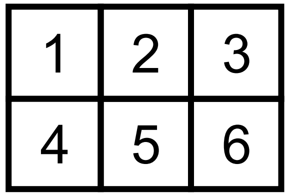


What is a NumPy array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                          order='F')  # F=Fortran
```



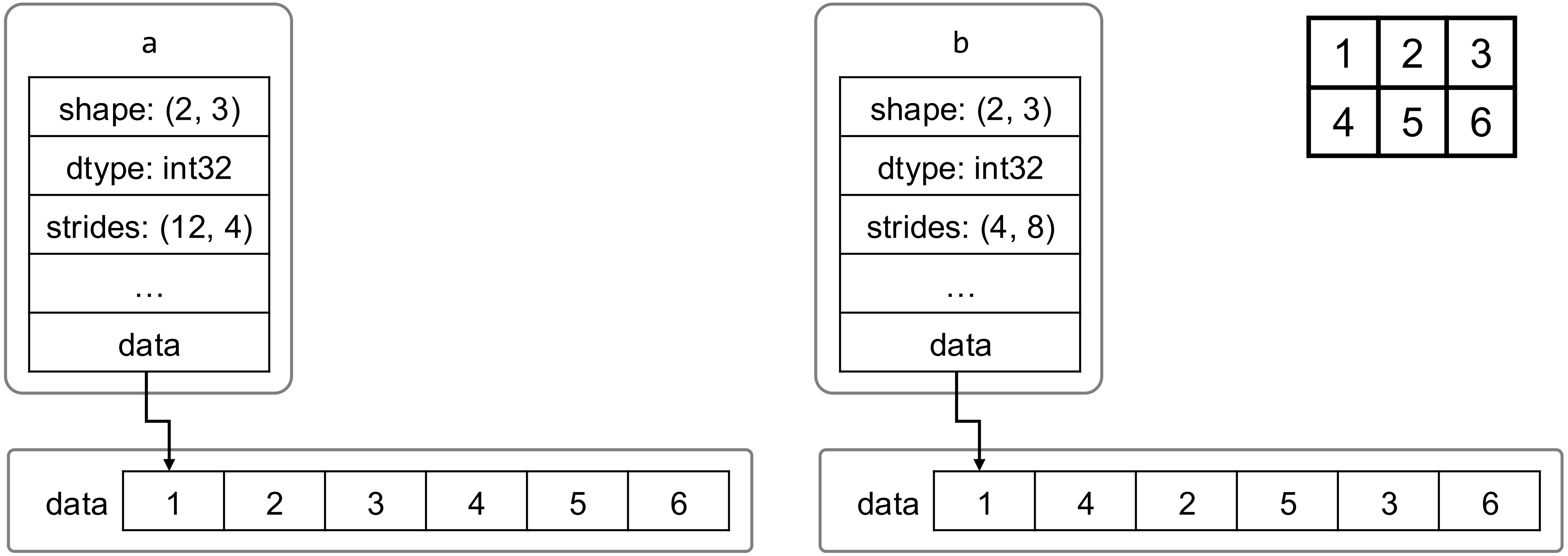




1	2	3
4	5	6

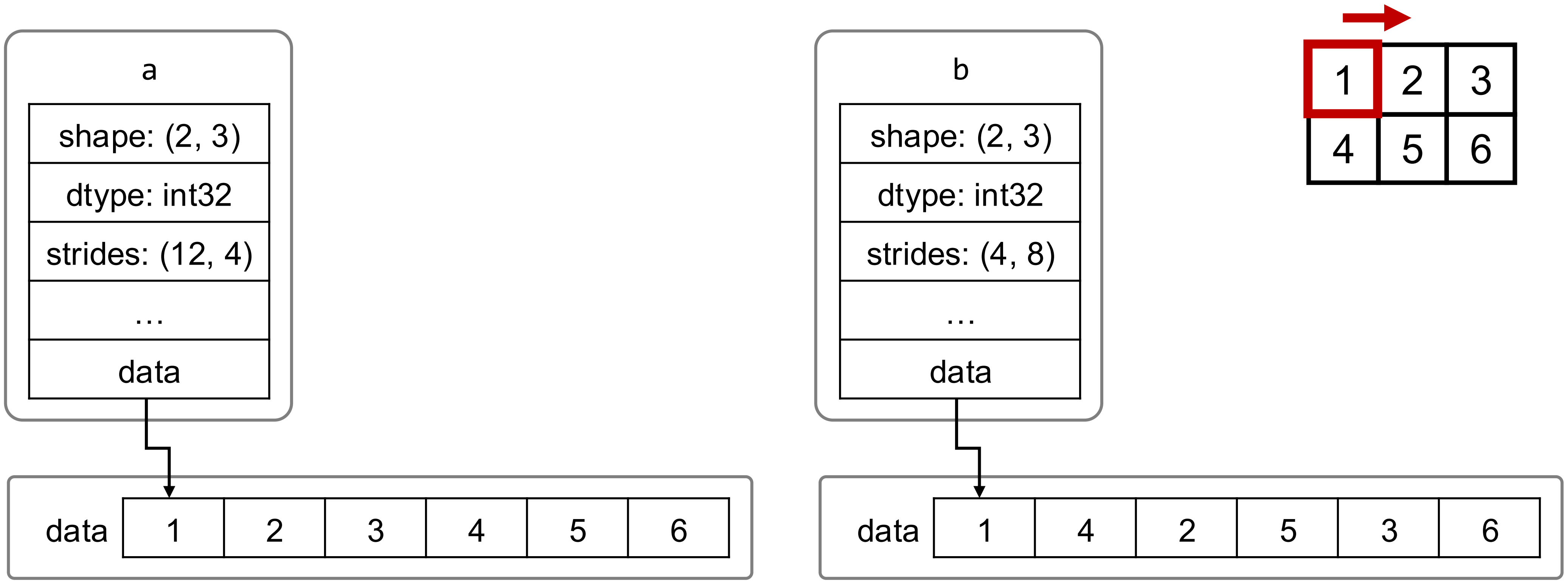
What is a NumPy array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                        order=‘F’)  # F=Fortran
```



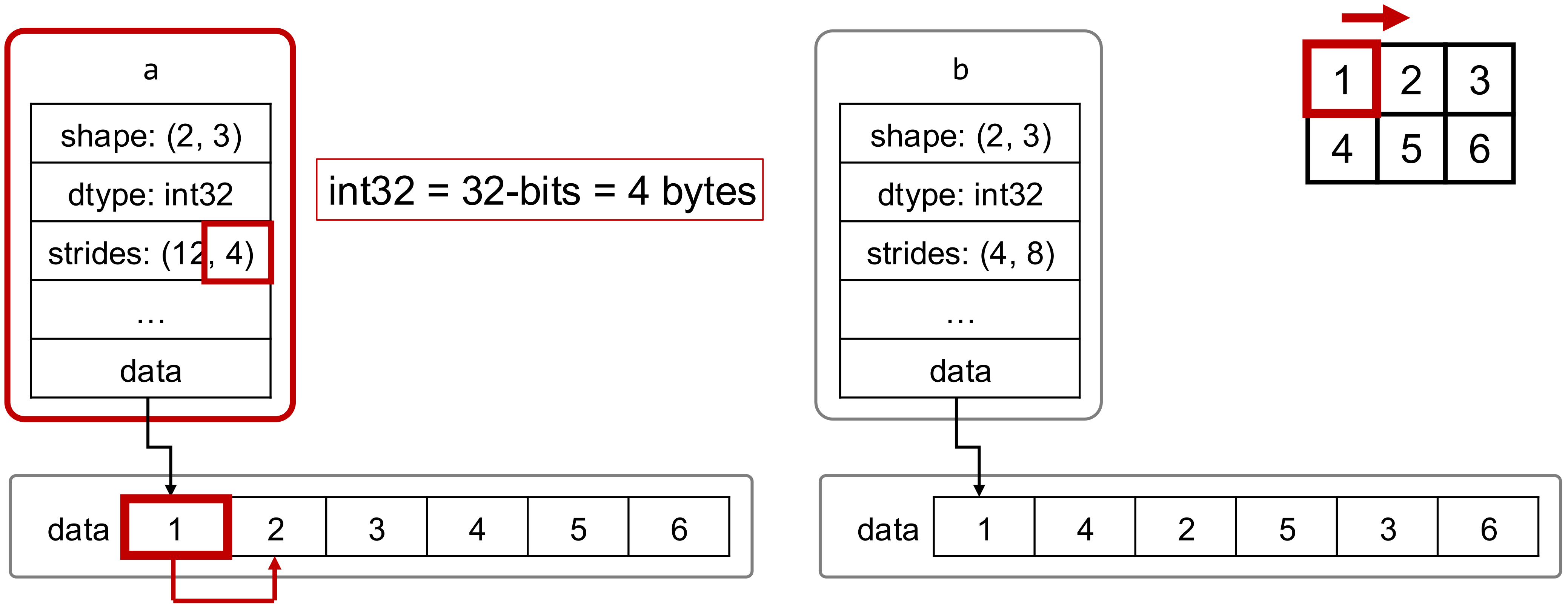
What is a NumPy array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                 order='F')  # F=Fortran
```



What is a NumPy array

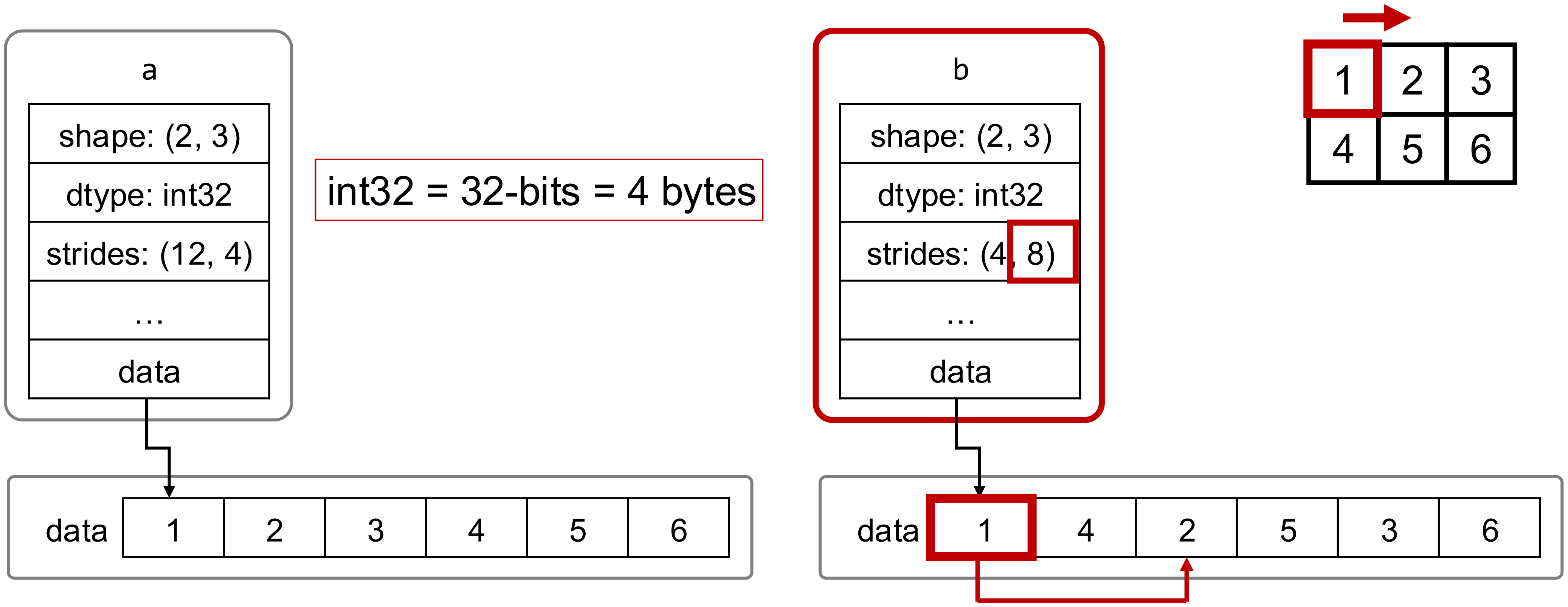
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                 order='F')  # F=Fortran
```



What is a NumPy array

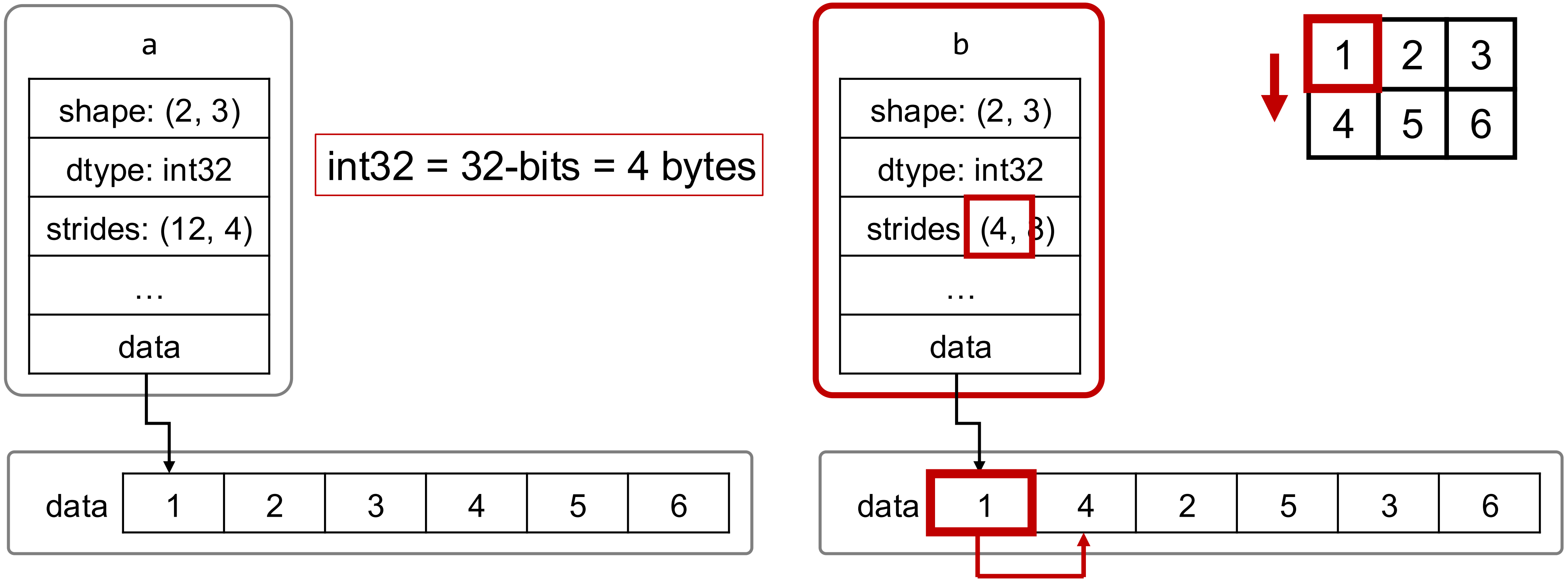
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                 order=‘F’)  # F=Fortran
```

int32 = 32-bits = 4 bytes



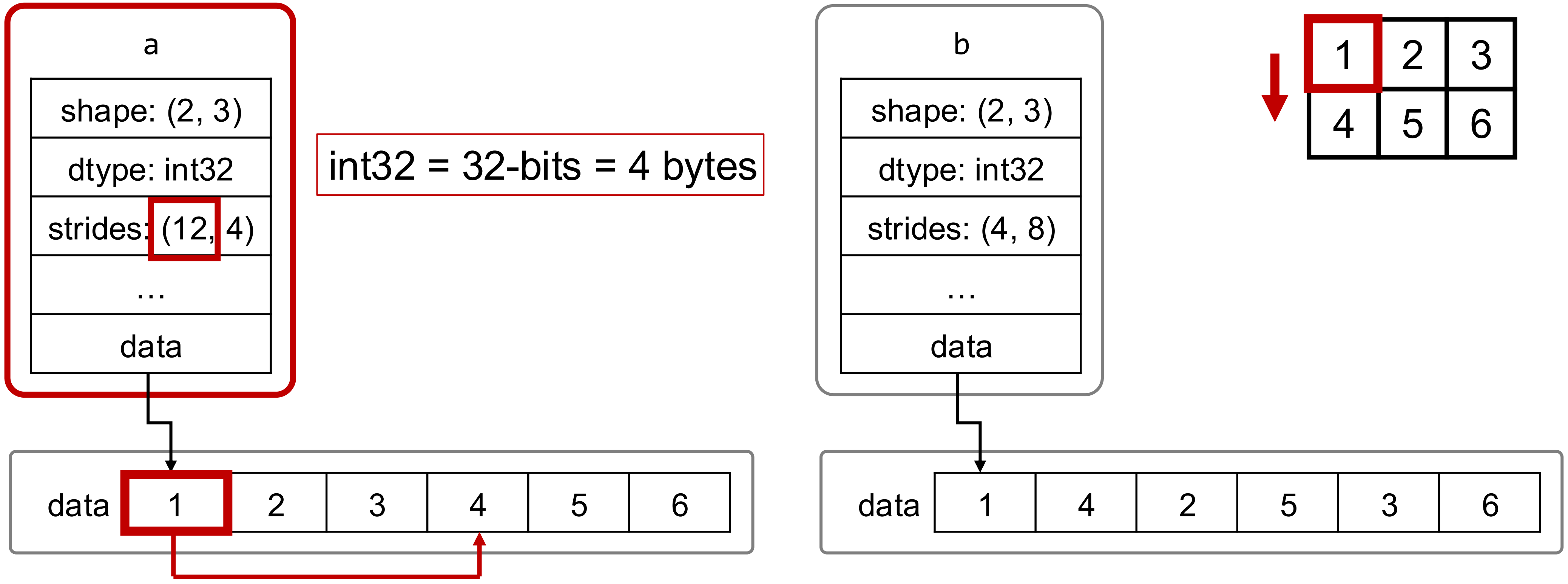
What is a NumPy array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                 order='F')  # F=Fortran
```



What is a NumPy array

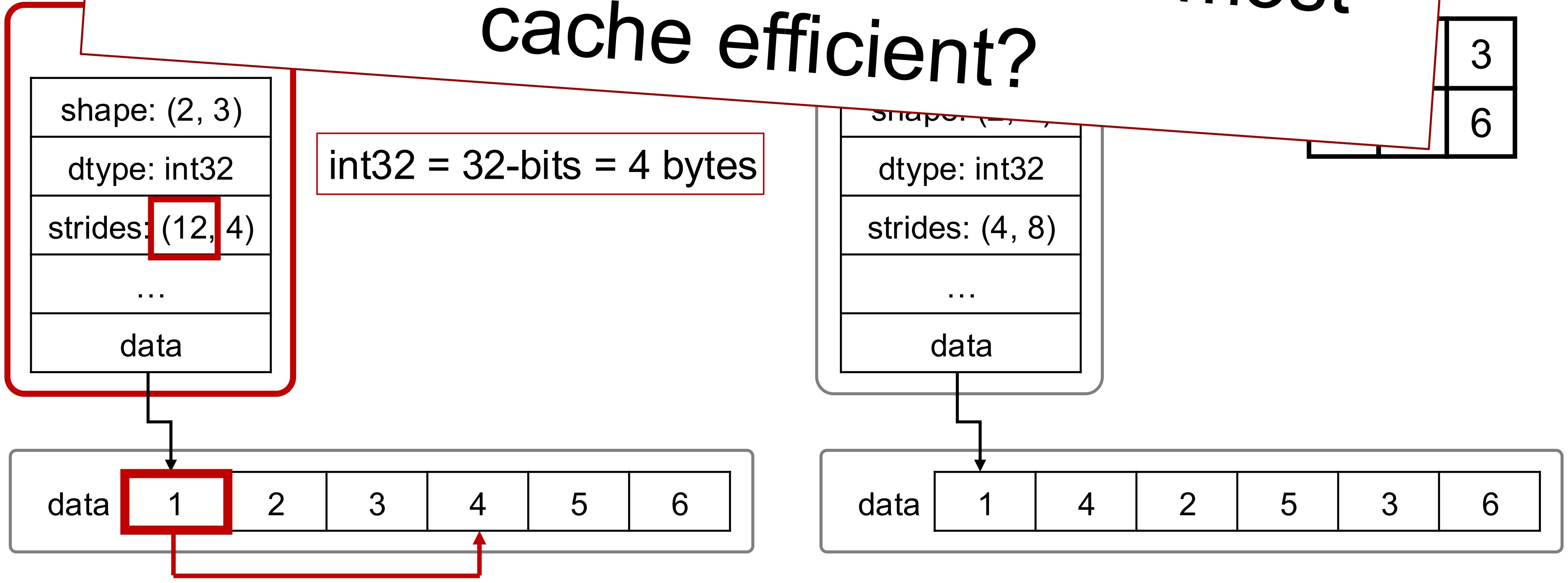
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([[1, 2, 3], [4, 5, 6]],
                           order='F')  # F=Fortran
```



What is a NumPy array

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

Question: what direction is most cache efficient?



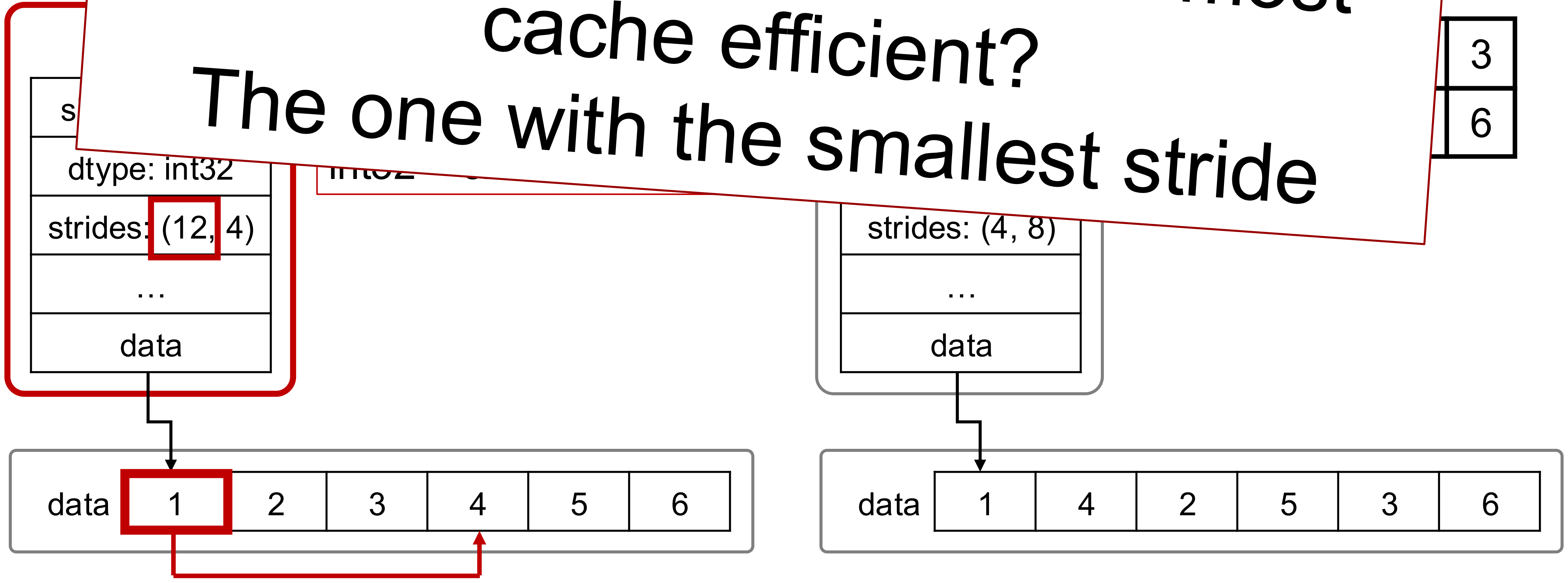
int32 = 32-bits = 4 bytes

What is a NumPy array

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

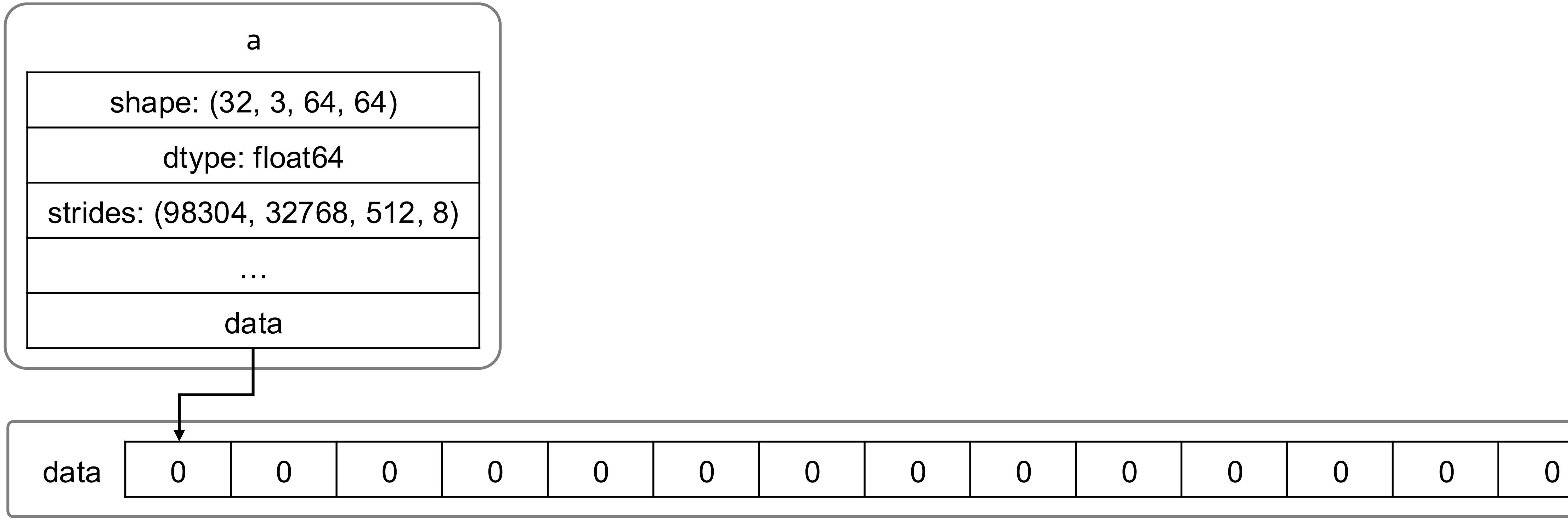
Question: what direction is most cache efficient?

The one with the smallest stride



What is a NumPy array

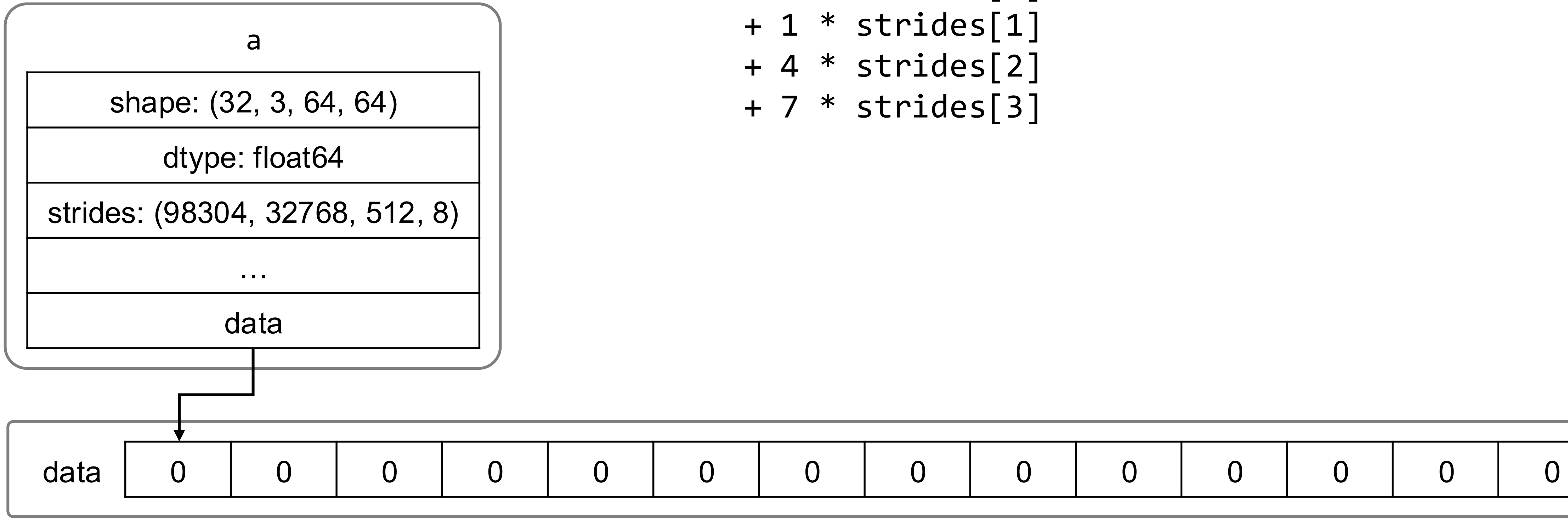
```
>>> a = np.zeros((32, 3, 64, 64))
```



What is a NumPy array

```
>>> a = np.zeros((32, 3, 64, 64))
>>> a[2, 1, 4, 7]
```

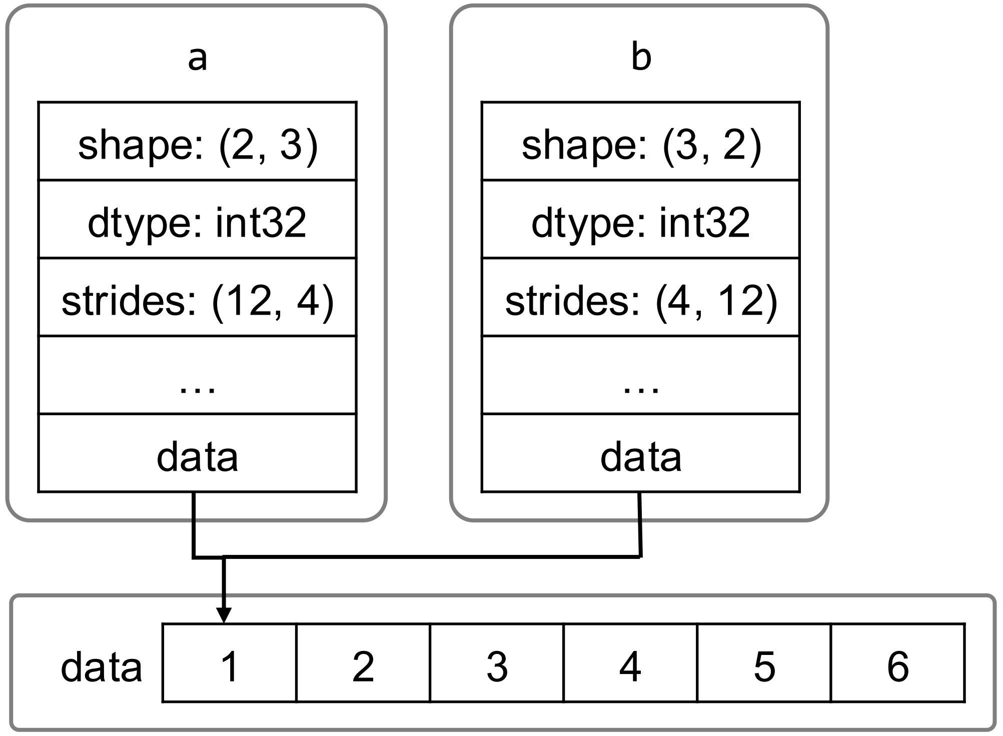
```
index = 2 * strides[0]
      + 1 * strides[1]
      + 4 * strides[2]
      + 7 * strides[3]
```

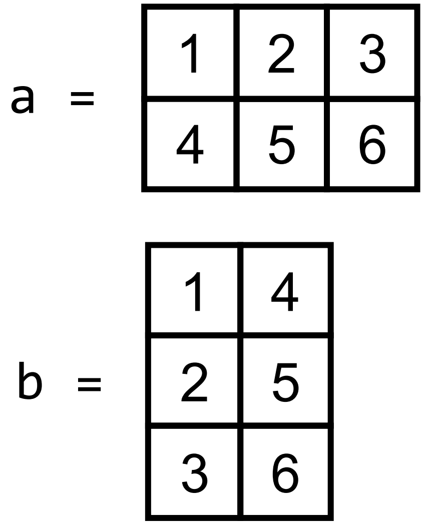


NumPy:
Views vs Copies

Views vs Copies

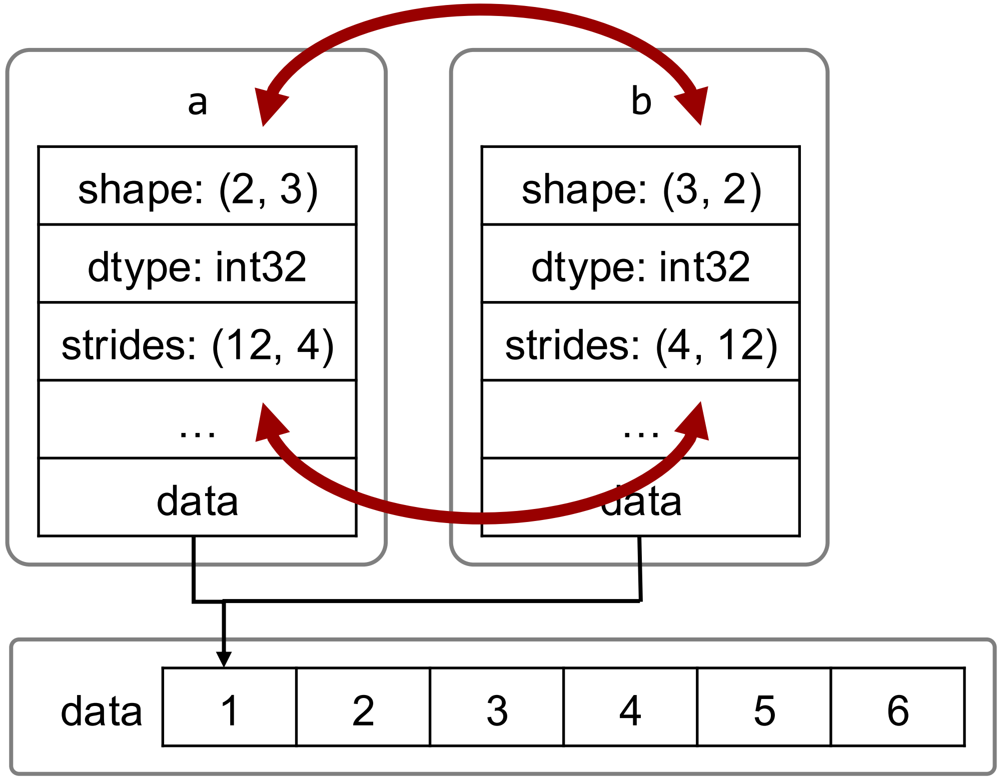
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.T  # Transpose
```





Views vs Copies

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.T  # Transpose
```

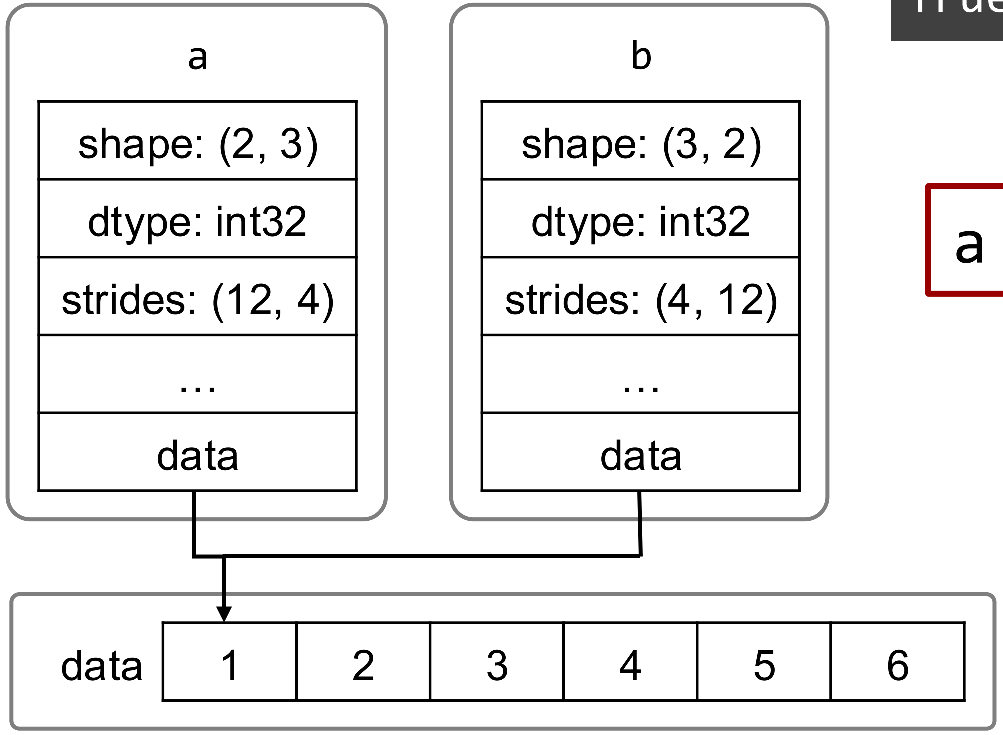


$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Views vs Copies

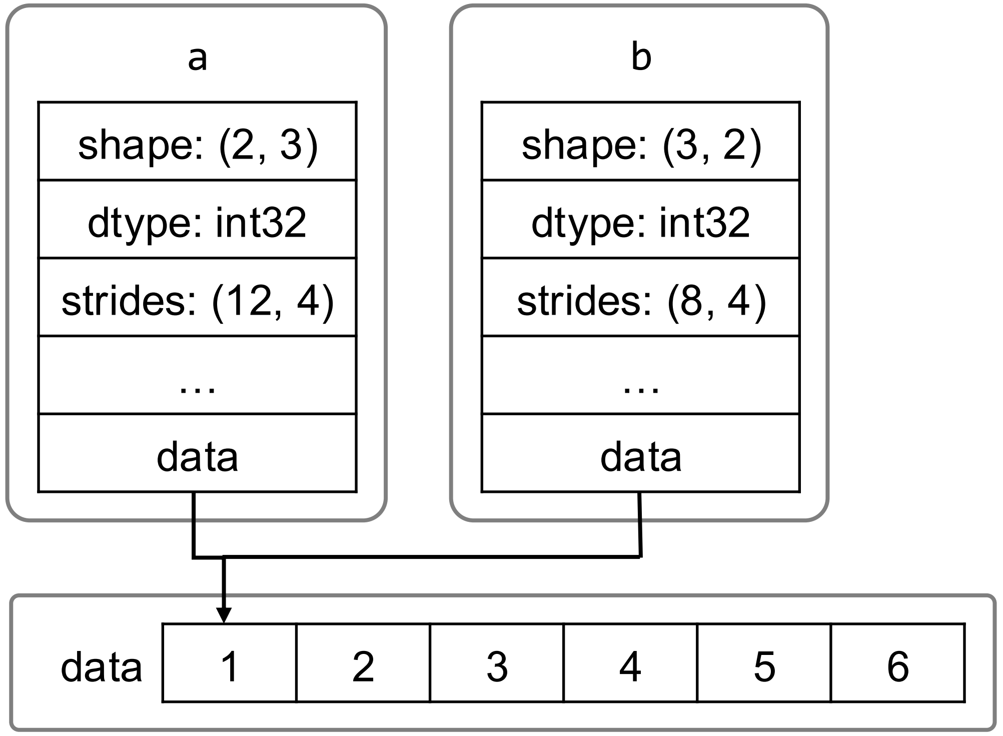
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.T  # Transpose
>>> np.shares_memory(a, b)
True
```



a and b are *views* to the same data

Views vs Copies

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

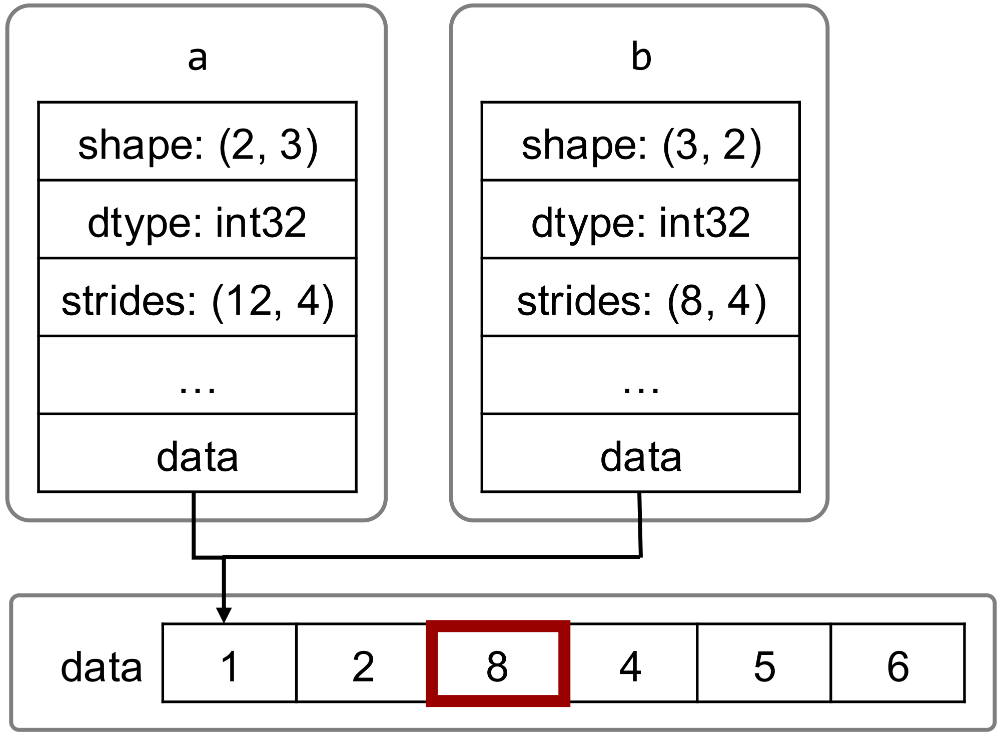


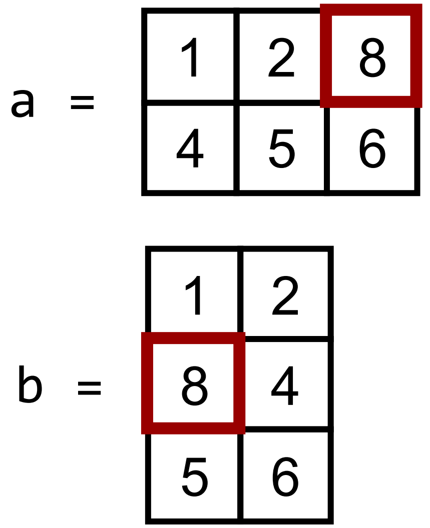
$$a = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline\end{array}$$

$$b = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline\end{array}$$

Views vs Copies

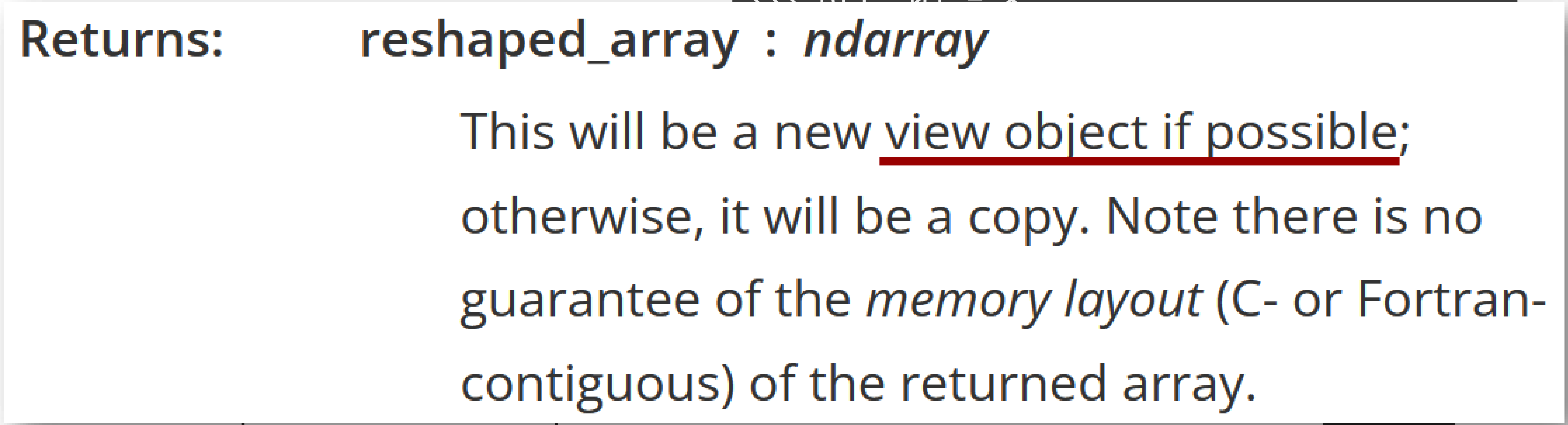
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.reshape(3, 2)
>>> b[1, 0] = 8
```





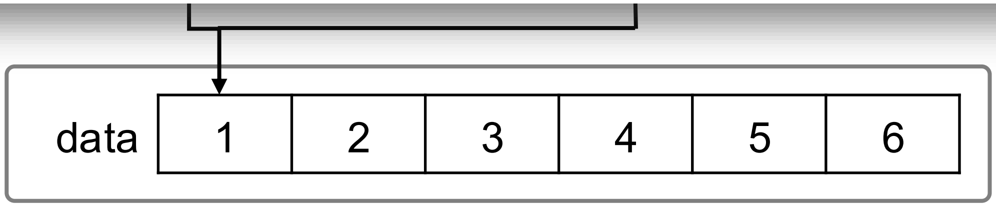
Views vs Copies

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.reshape(3, 2)
>>> b[1, 0] = 8
```



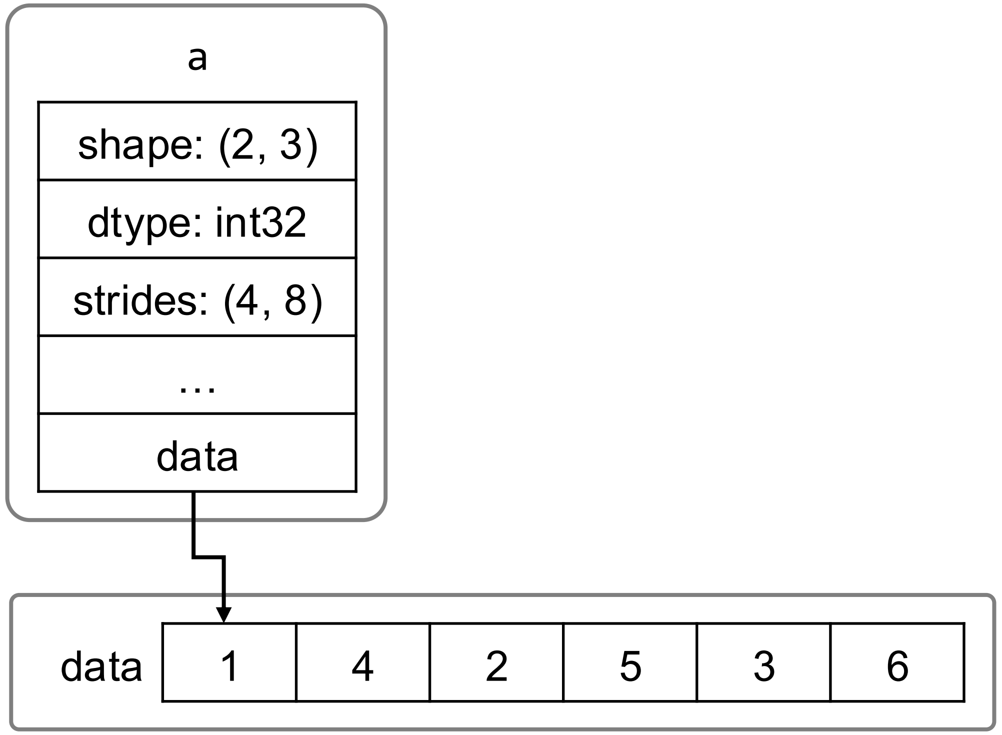
Returns: reshaped_array : *ndarray*

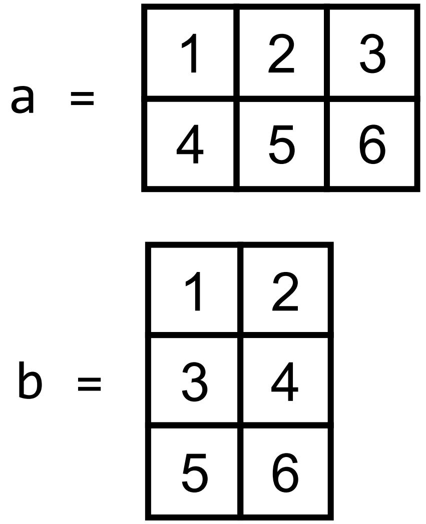
This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the *memory layout* (C- or Fortran- contiguous) of the returned array.



Views vs Copies

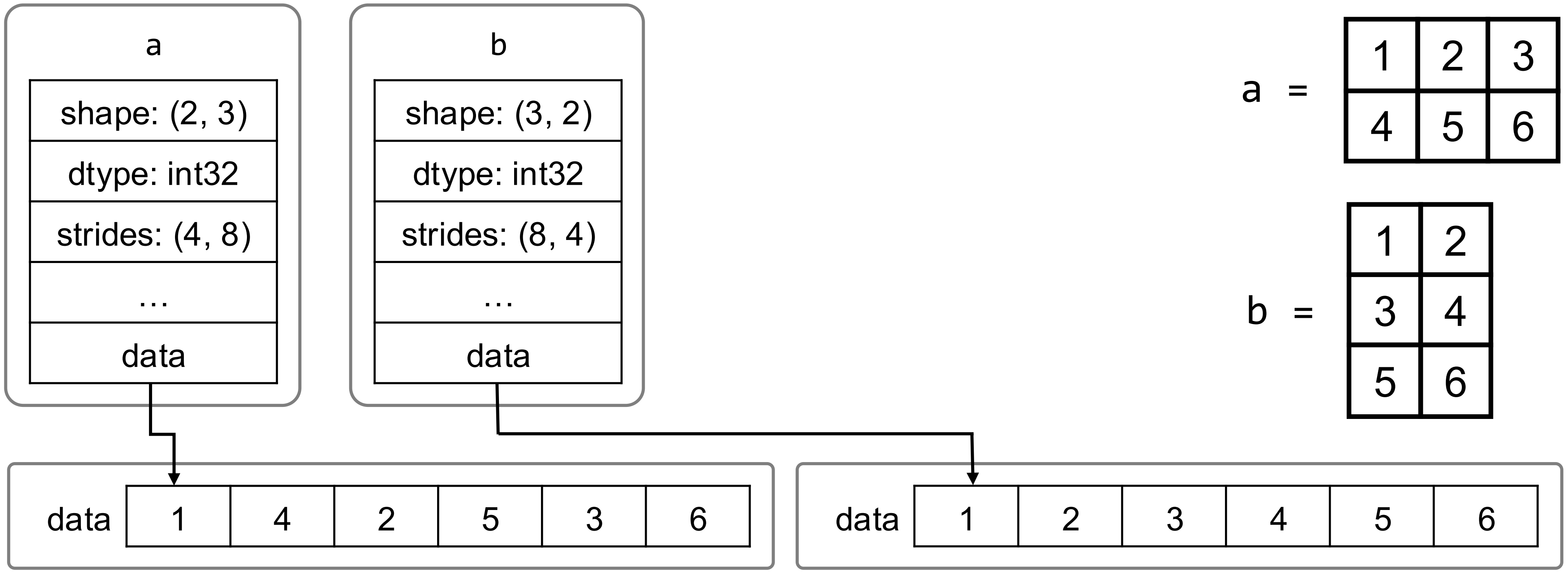
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]],
                 order=‘F’)  # Fortran
>>> b = a.reshape(3, 2)
```





Views vs Copies

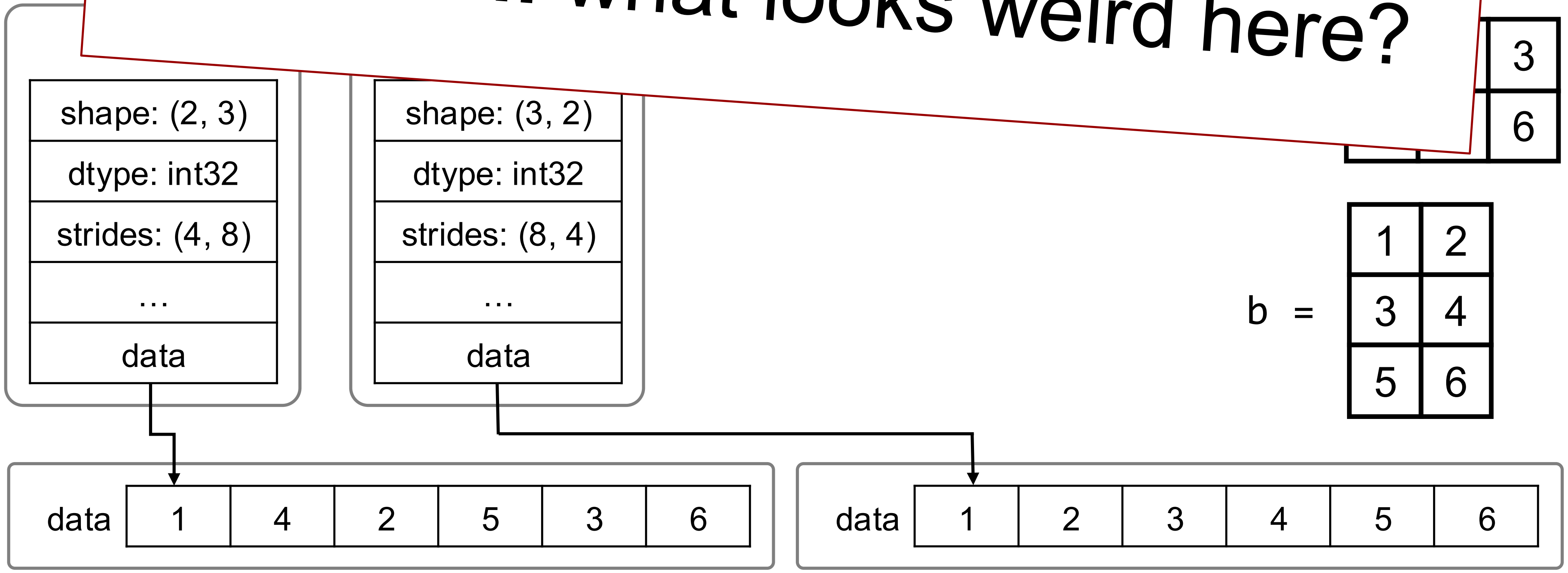
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]],
                  order='F')  # Fortran
>>> b = a.reshape(3, 2)
```



Views vs Copies

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]],
                 order='F')  # Fortran
```

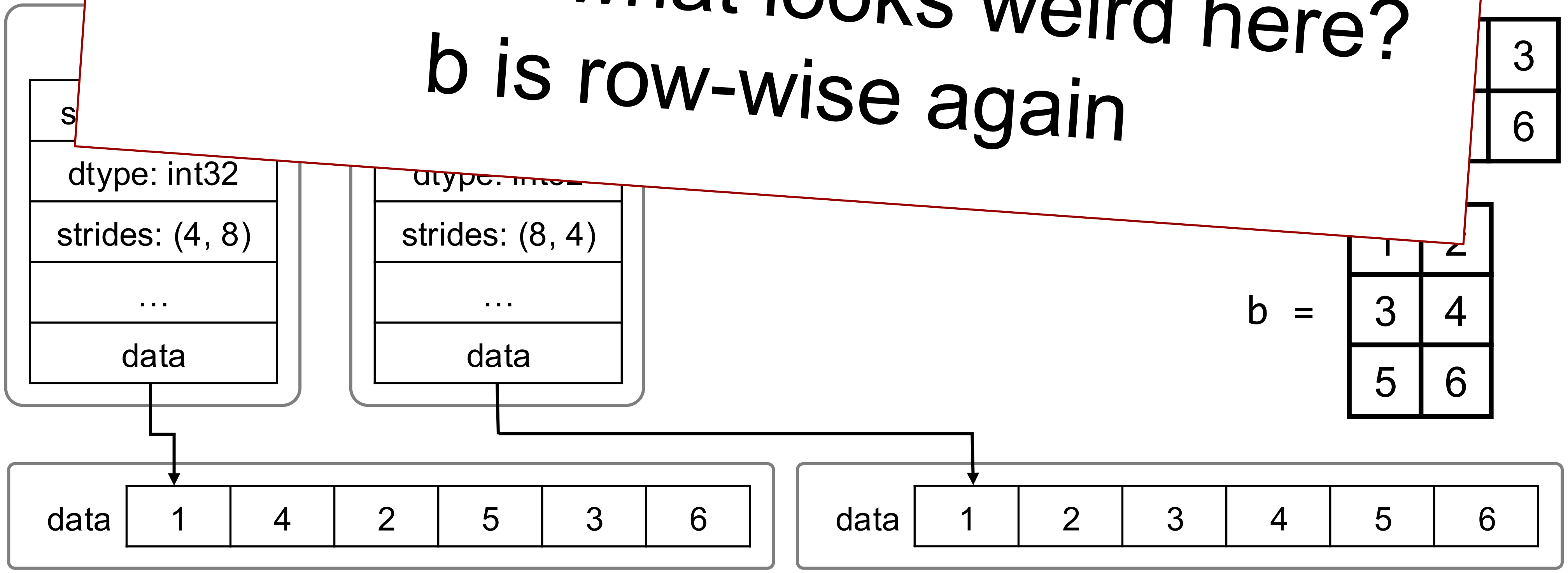
Question: what looks weird here?



Views vs Copies

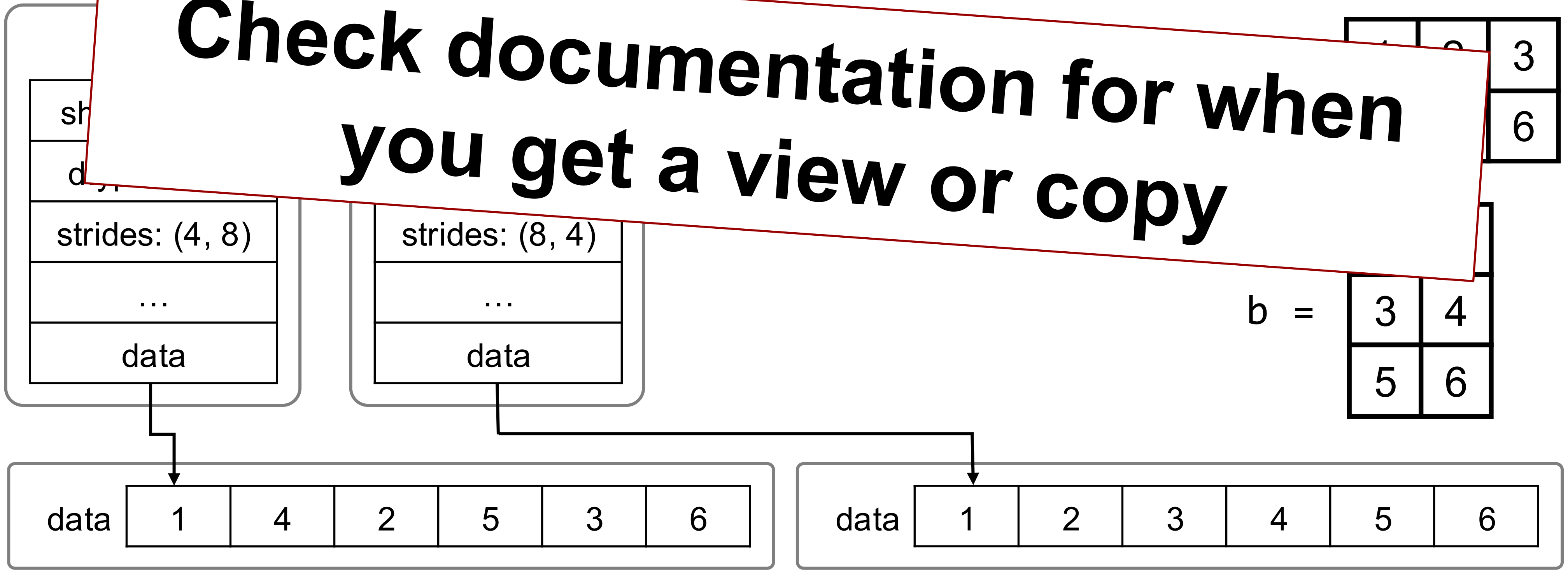
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]],
                 order='F')  # Fortran
```

Question: what looks weird here?
b is row-wise again



Views vs Copies

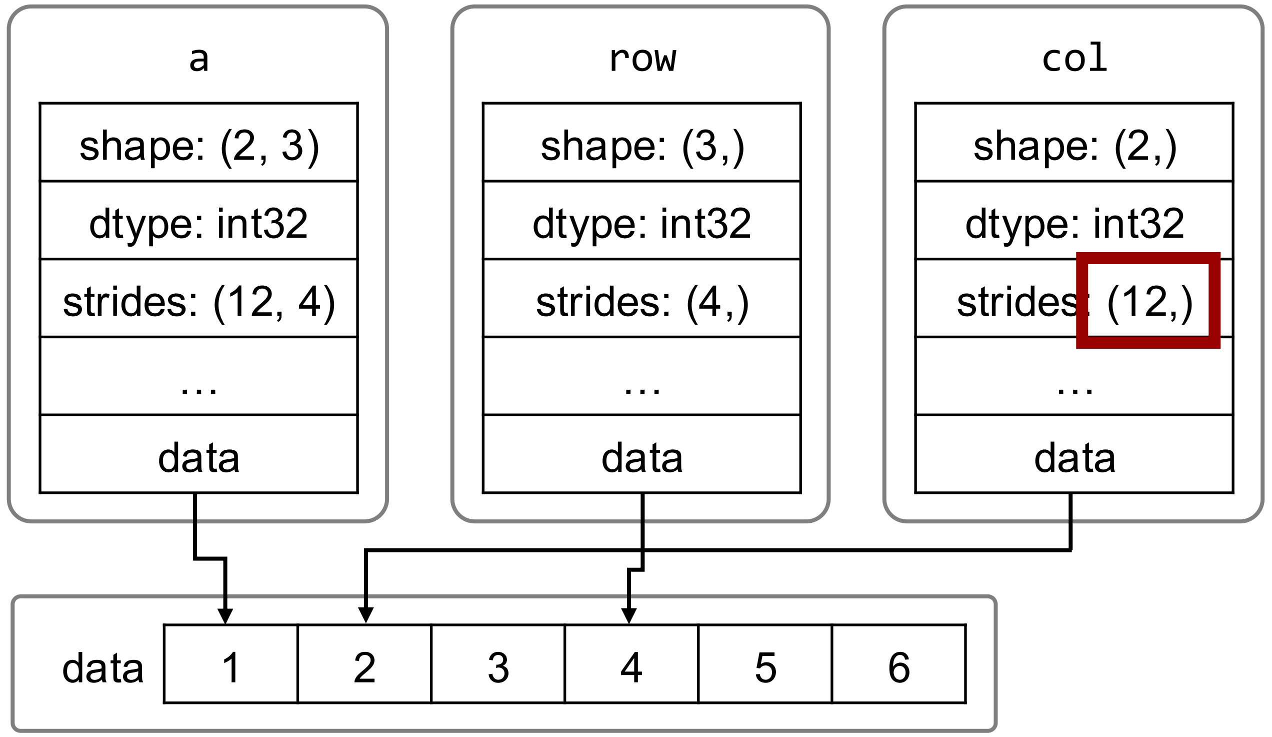
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]],
                 order='F')  # Fortran
>>> b = a.reshape(3, 2)
```



Check documentation for when you get a view or copy

Views vs Copies: Indexing

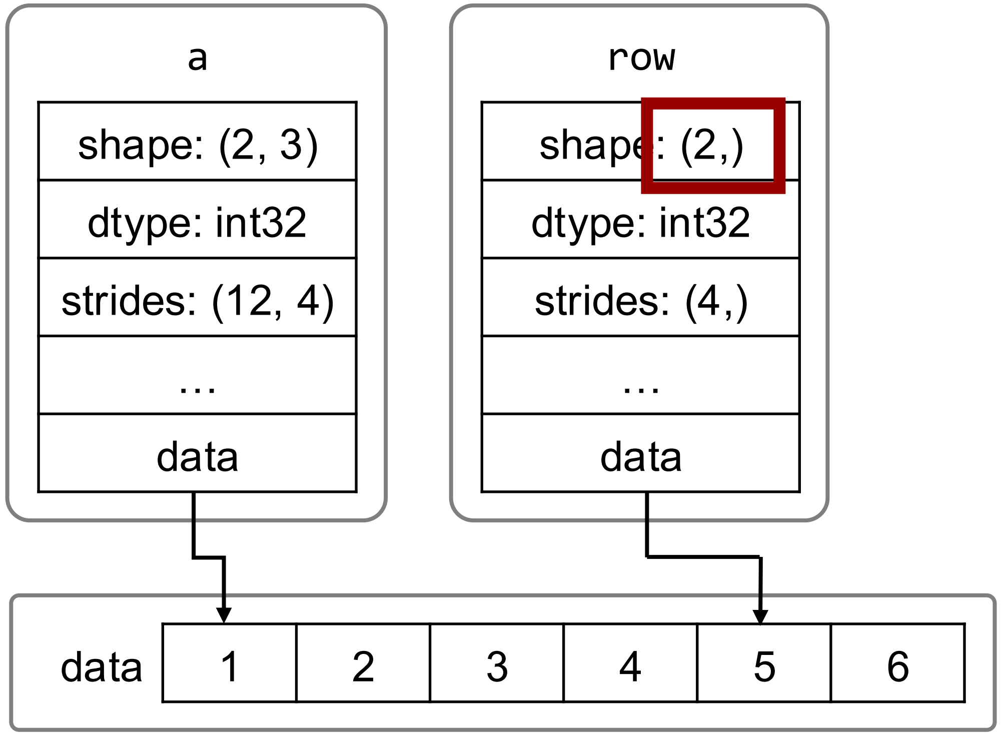
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> row, col = a[1, :], a[:, 1]
```



$$a = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array}$$

Views vs Copies: Indexing

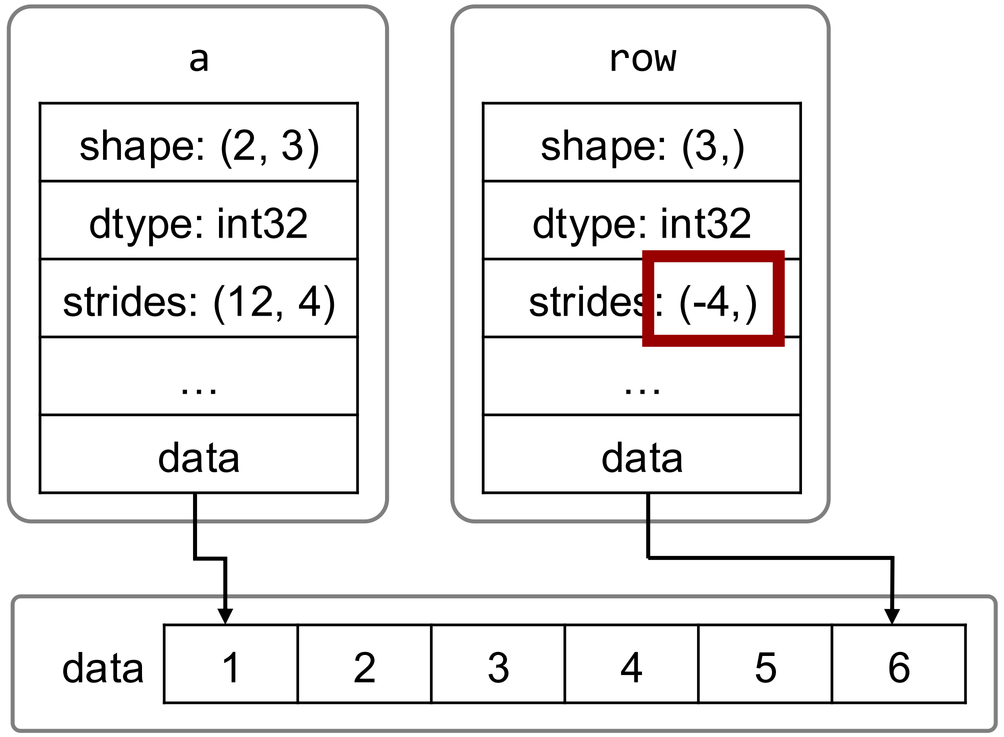
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> row = a[1, 1:]; row
array([5, 6])
```



$$a = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array}$$

Views vs Copies: Indexing

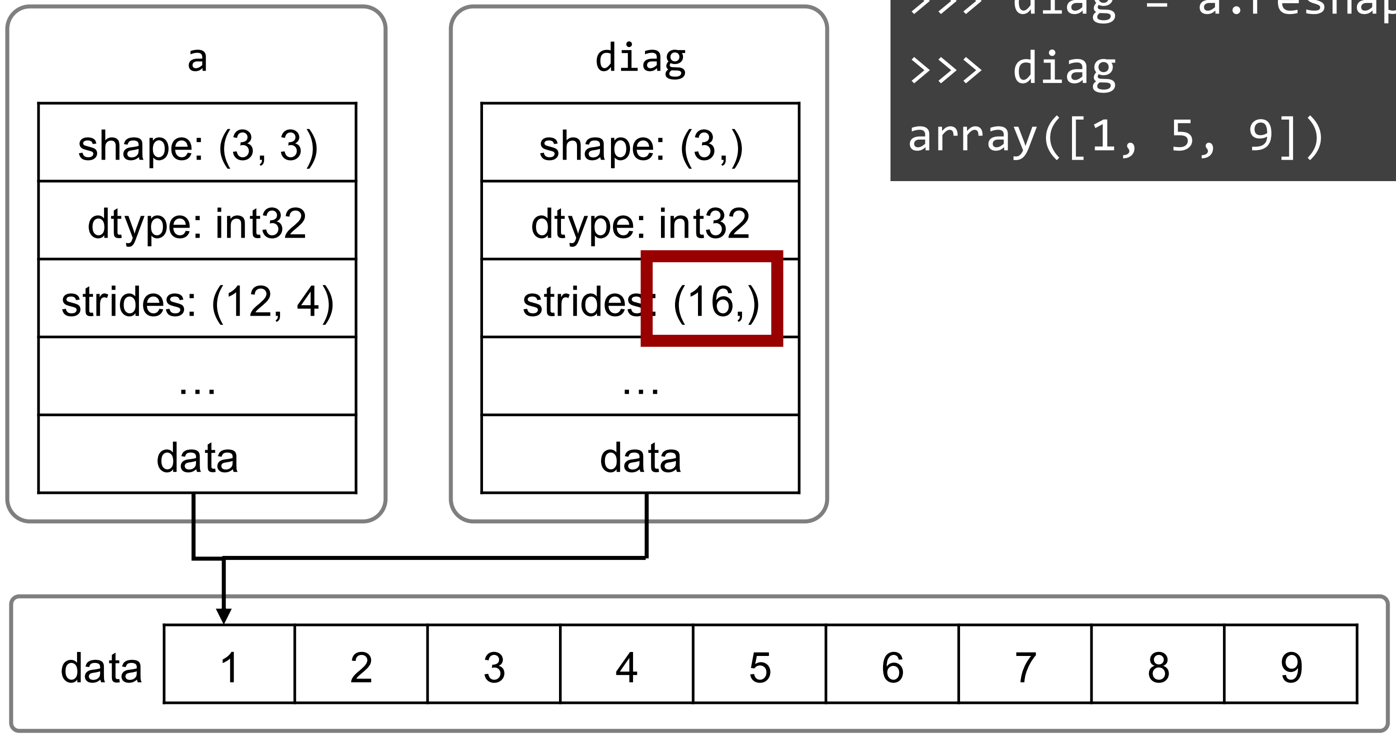
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> row = a[1, ::-1]; row
array([6, 5, 4])
```

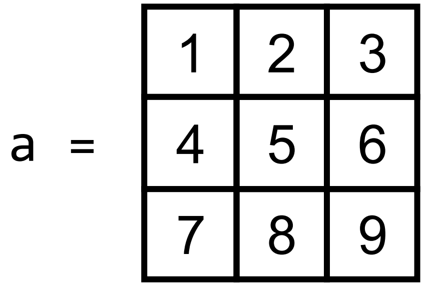


$$a = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline\end{array}$$

Views vs Copies: Indexing

```
>>> a = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
>>> diag = a.reshape(-1)[::4]
>>> diag
array([1, 5, 9])
```



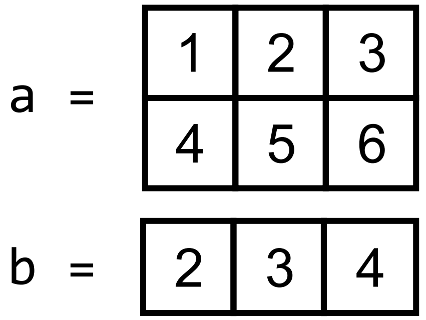


Quiz time!

NumPy: Broadcasting

Broadcasting

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([2, 3, 4])
>>> a + b
array([[ 3,  5,  7],
       [ 6,  8, 10]])
```



Broadcasting

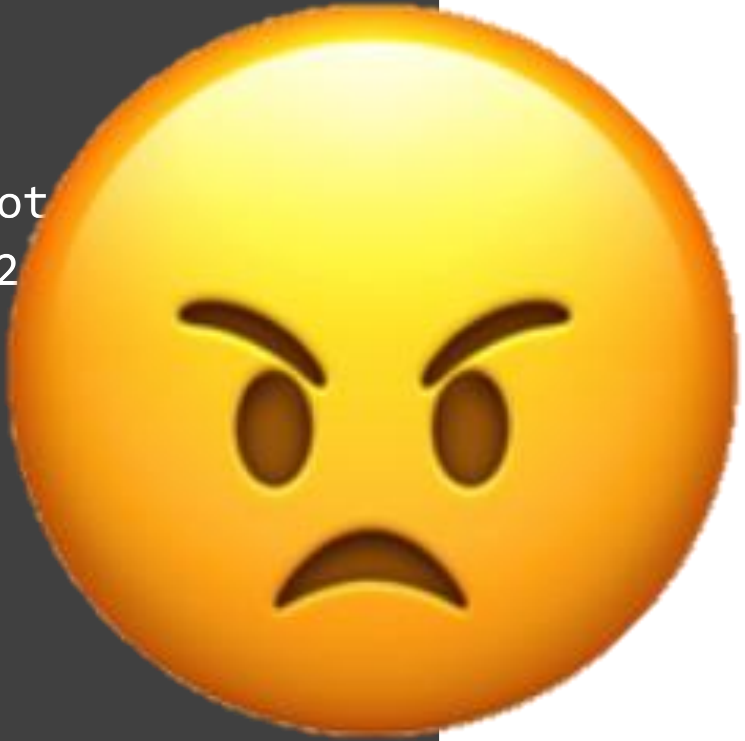
```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([2, 3])
>>> a + b
...
ValueError: operands could not be broadcast
together with shapes (2,3) (2,)
```

$$a = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array}$$

$$b = \begin{array}{|c|c|}\hline 2 & 3 \\ \hline \end{array}$$

Broadcasting

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([2, 3])
>>> a + b
...
ValueError: operands could not
together with shapes (2,3) (2
```



a =

1	2	3
4	5	6

b =

2	3

Broadcasting

```
>>> a.shape
(2, 3,)
>>> b.shape
(3,)
```

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
       (3,)
```

1. Line up the shapes to the right

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
    (1, 3,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
    (2, 3,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
    (2, 3,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(2, 3,)
>>> b.shape
(2,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
       (2,)
```

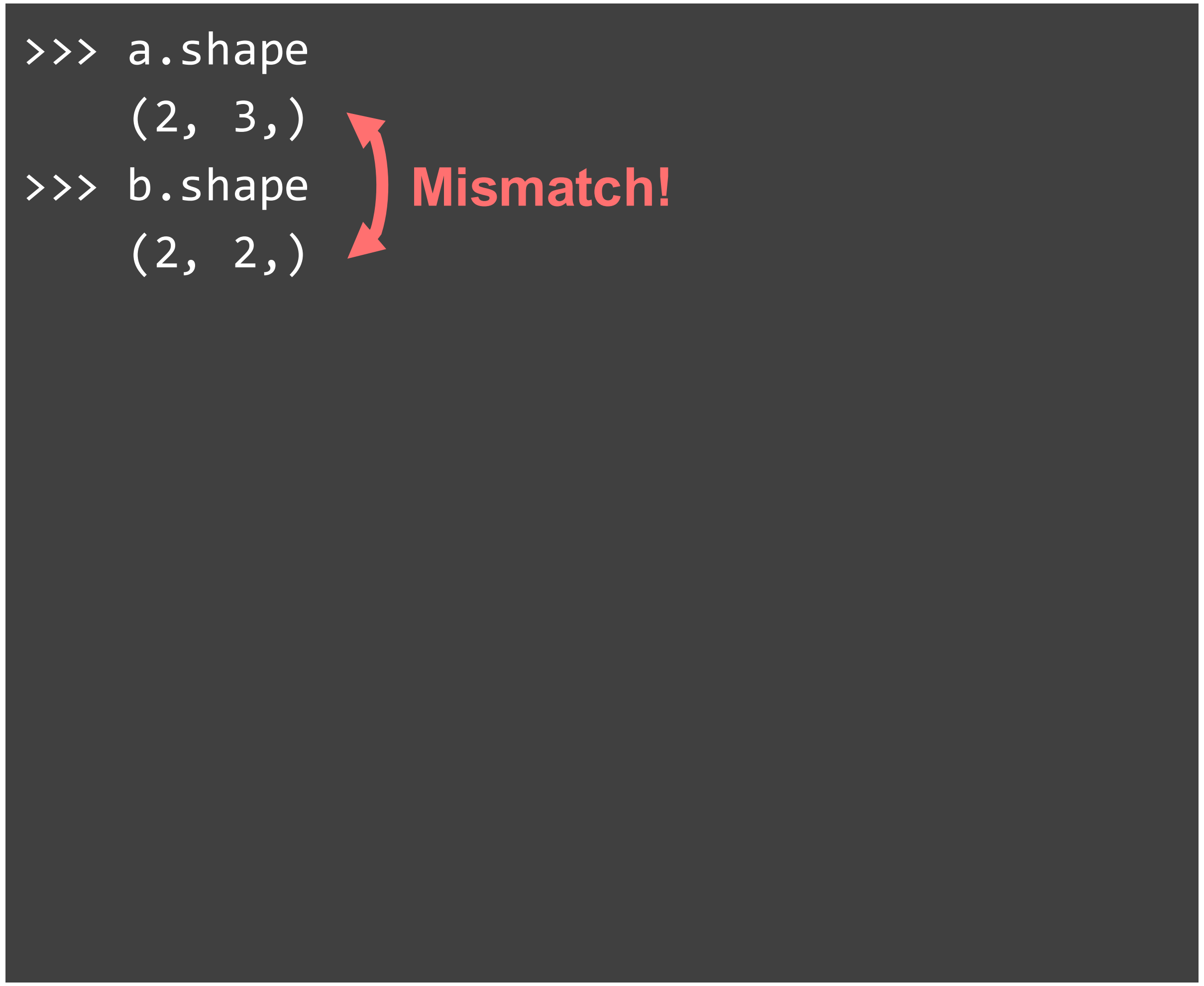
1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
    (1, 2,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting



```
>>> a.shape
    (2, 3,)
>>> b.shape
    (2, 2,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(2, 3,)
>>> b.shape
(2,)
>>> b = b[:, None]  # Add dimension
>>> b.shape
(2, 1,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
(2,)
>>> b = b[:, None]  # Add dimension
>>> b.shape
    (2, 1,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
    (2, 3,)
>>> b.shape
(2,)
>>> b = b[:, None]  # Add dimension
>>> b.shape
    (2, 3,)
```

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
   N, C,  W,  H
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
>>> a - m
...
ValueError: operands could not be broadcast
together with shapes (128,3,64,64) (3,)
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
             (3,)
>>> a - m
...
ValueError: operands could not be broadcast
together with shapes (128,3,64,64) (3,)
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(  1, 1,  1,  3,)
>>> a - m
...
ValueError: operands could not be broadcast
together with shapes (128,3,64,64) (3,)
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(128, 3, 64,  3,)
>>> a - m
...
ValueError: operands could not be broadcast
together with shapes (128,3,64,64) (3,)
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
>>> m = m[:, None, None]  # Add dimensions
>>> m.shape
(3, 1, 1,)
>>> a - m
>>>
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
>>> m = m[:, None, None]  # Add dimensions
>>> m.shape
     (3,  1,  1,)
>>> a - m
>>>
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
>>> m = m[:, None, None]  # Add dimensions
>>> m.shape
  (1, 3,  1,  1,)
>>> a - m
>>>
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
>>> m = m[:, None, None]  # Add dimensions
>>> m.shape
(128, 3, 64, 64,)
>>> a - m
>>>
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(128, 3, 64, 64,)
>>> m.shape
(3,)
>>> m = m[:, None, None]  # Add dimensions
>>> m.shape
(3, 1, 1,)
>>> a – m
>>> m[None].shape
(1, 3, 1, 1,)
```

Stack of 128 RGB images of shape 64 x 64

1. Line up the shapes to the right
2. Left-pad the shortest with 1s
3. If an element is 1, replace with larger
4. If all elements match: broadcasted!

Broadcasting

```
>>> a.shape
(2, 3,)
>>> b.shape
(3,)
```

Broadcasting

```
>>> a.shape
(2, 3,)
>>> b.shape
(3,)
>>> a, b = np.broadcast_arrays(a, b)
>>> print(f”a: {a.shape}, b: {b.shape}”)
a: (2, 3), b: (2, 3)
```

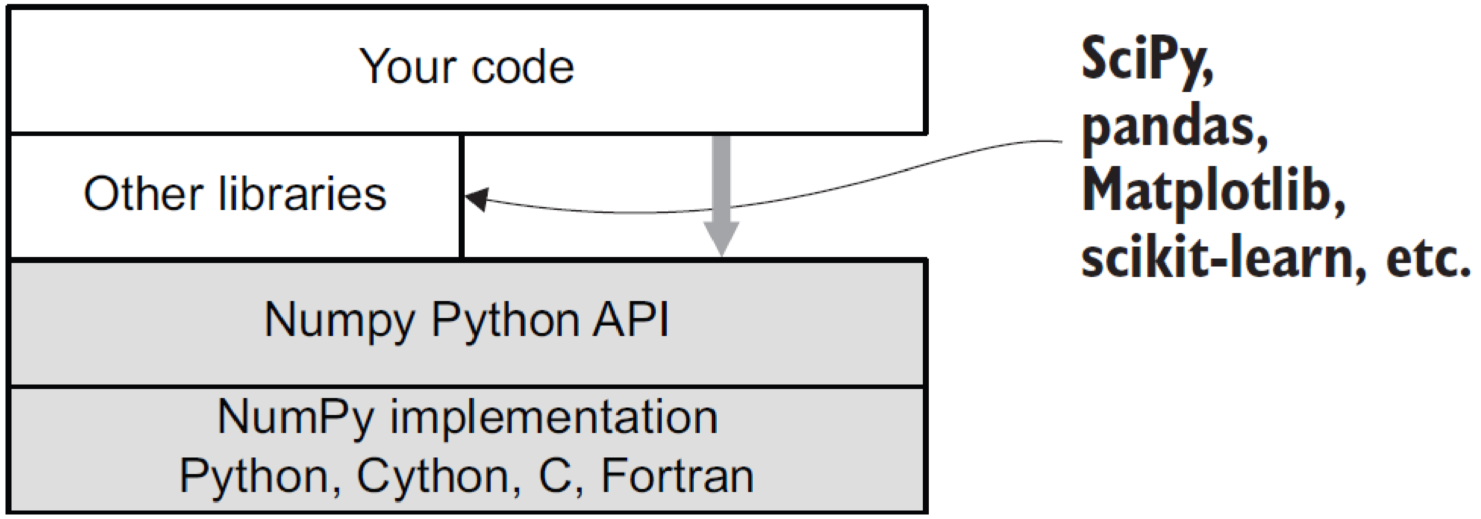
Broadcasting

```
>>> a.shape
(2, 3,)
>>> b.shape
(3,)
>>> a, b = np.broadcast_arrays(a, b)
>>> print(f”a: {a.shape}, b: {b.shape}”)
a: (2, 3), b: (2, 3)
>>> print(f”a: {a.strides}, b: {b.strides}”)
a: (24, 8), b: (0, 8)
```

Quiz time!

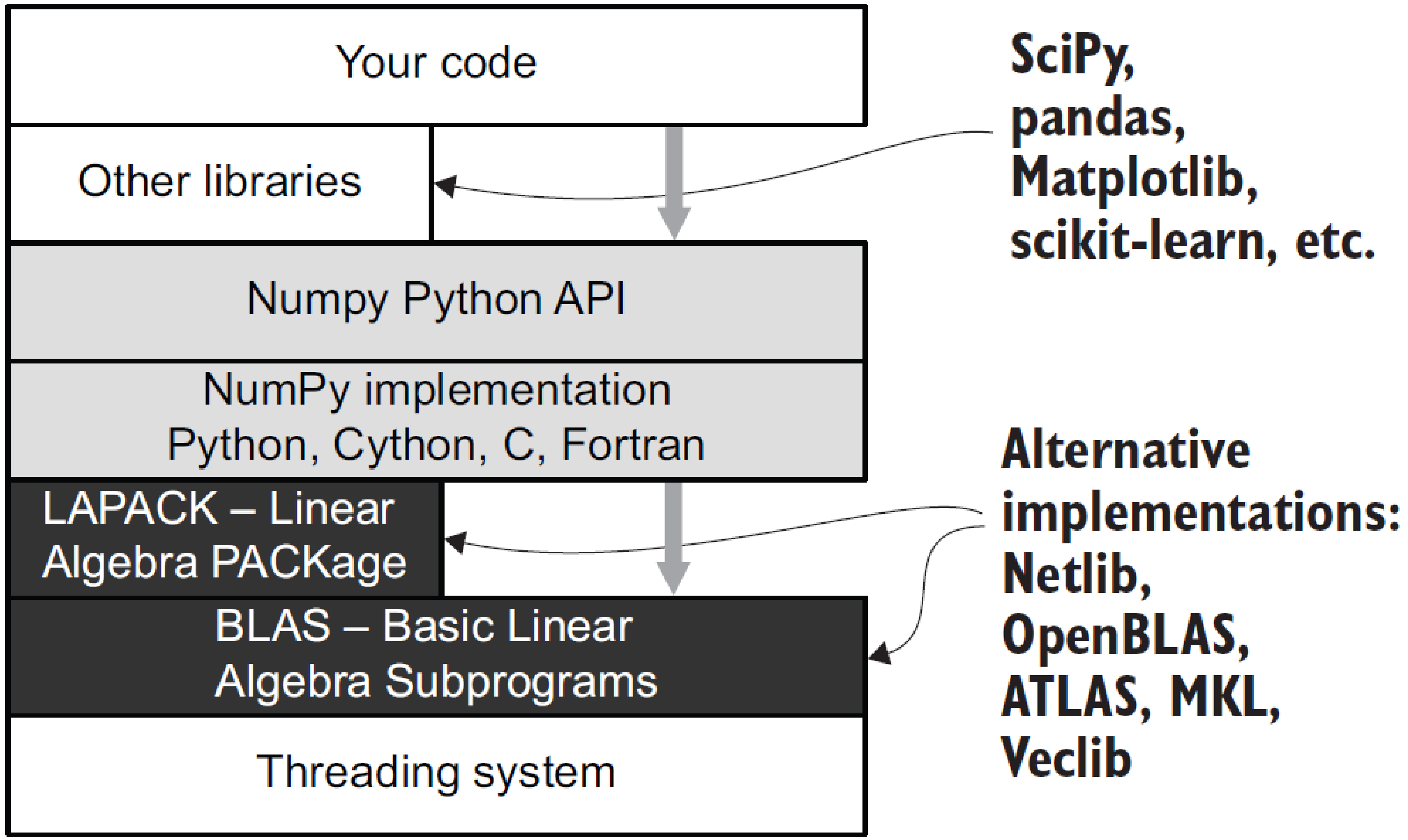
NumPy Backends: The foundation of the foundation

NumPy backends



Fast Python - Figure 4.11

NumPy backends


Your code
Other libraries
Numpy Python API
NumPy implementation
Python, Cython, C, Fortran
LAPACK – Linear Algebra PACKage
BLAS – Basic Linear Algebra Subprograms
Threading system
SciPy, pandas, Matplotlib, scikit-learn, etc.
Alternative implementations: Netlib, OpenBLAS, ATLAS, MKL, Veclib

Fast Python - Figure 4.11

NumPy backends

```
$ conda install numpy
```

```
$ pip install numpy
```

```
$ conda install -c conda-forge numpy
```

NumPy backends

```
$ conda install numpy
$ python
>>> import numpy as np
>>> np.show_config()
Build Dependencies:
  blas:
    detection method: pkgconfig
    found: true
    include directory: …
    lib directory:
    name: mkl-sdl
    openblas configuration: unknown
…
```

```
$ pip install numpy
$ python
>>> import numpy as np
>>> np.show_config()
…
"Build Dependencies": {
    "blas": {
      "name": "openblas64",
      "found": true,
      "version": "0.3.23.dev",
      "detection method": "pkgconfig",
      "include directory": …,
      "lib directory": "/usr/local/lib",
      "openblas configuration": …
…
```

```
$ conda install -c conda-forge numpy
$ python
>>> import numpy as np
>>> np.show_config()
…
"Build Dependencies": {
    "blas": {
      "name": "blas",
      "found": true,
      "version": "3.9.0",
      "detection method": "pkgconfig",
      "include directory": …,
      "lib directory": …,
      "openblas configuration": …
…
```

NumPy backends: A Laptop

```
$ conda install numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
1.76 s.

Dotted two vectors of length 524288 in:
0.28 ms.

SVD of a 2048x1024 matrix in:
0.72 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.17 s.

Eigendecomposition of a 2048x2048
matrix in:
6.77 s.
```

```
$ pip install numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
1.49 s.

Dotted two vectors of length 524288 in:
0.29 ms.

SVD of a 2048x1024 matrix in:
1.48 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.19 s.

Eigendecomposition of a 2048x2048
matrix in:
8.01 s.
```

```
$ conda install -c conda-forge numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
1.76 s.

Dotted two vectors of length 524288 in:
0.30 ms.

SVD of a 2048x1024 matrix in:
0.77 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.18 s.

Eigendecomposition of a 2048x2048
matrix in:
6.92 s.
```

https://web.archive.org/web/20230130105308/https://markus-beuckelmann.de/blog/boosting-numpy-blas.html

NumPy backends: HPC with XeonGold6226R

```
$ conda install numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
2.02 s.

Dotted two vectors of length 524288 in:
0.35 ms.

SVD of a 2048x1024 matrix in:
1.11 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.30 s.

Eigendecomposition of a 2048x2048
matrix in:
9.23 s.
```

```
$ pip install numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
2.00 s.

Dotted two vectors of length 524288 in:
0.37 ms.

SVD of a 2048x1024 matrix in:
1.24 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.35 s.

Eigendecomposition of a 2048x2048
matrix in:
8.70 s.
```

```
$ conda install -c conda-forge numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
1.97 s.

Dotted two vectors of length 524288 in:
0.36 ms.

SVD of a 2048x1024 matrix in:
1.12 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.32 s.

Eigendecomposition of a 2048x2048
matrix in:
8.48 s.
```

https://web.archive.org/web/20230130105308/https://markus-beuckelmann.de/blog/boosting-numpy-blas.html

NumPy backends: HPC with XeonE5_2650v4

```
$ conda install numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
4.02 s.

Dotted two vectors of length 524288 in:
0.27 ms.

SVD of a 2048x1024 matrix in:
1.30 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.22 s.

Eigendecomposition of a 2048x2048
matrix in:
10.32 s.
```

```
$ pip install numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
4.59 s.

Dotted two vectors of length 524288 in:
0.27 ms.

SVD of a 2048x1024 matrix in:
1.46 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.22 s.

Eigendecomposition of a 2048x2048
matrix in:
10.81 s.
```

```
$ conda install -c conda-forge numpy
$ python numpy_bench.py
Dotted two 4096x4096 matrices in:
4.69 s.

Dotted two vectors of length 524288 in:
0.28 ms.

SVD of a 2048x1024 matrix in:
1.62 s.

Cholesky decomposition of a
2048x2048 matrix in:
0.24 s.

Eigendecomposition of a 2048x2048
matrix in:
12.83 s.
```

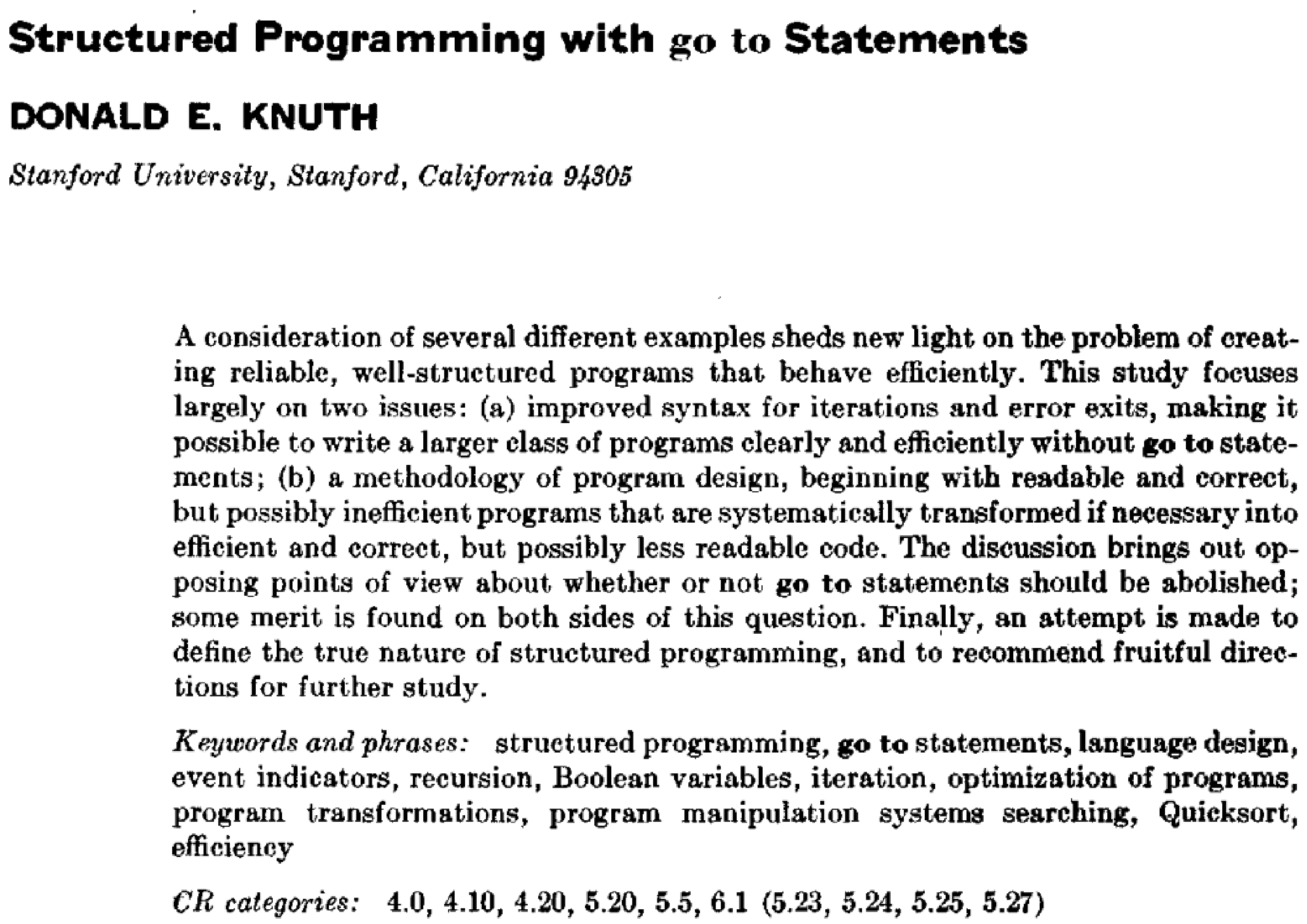
https://web.archive.org/web/20230130105308/https://markus-beuckelmann.de/blog/boosting-numpy-blas.html

Profiling

Profiling

”Premature optimization is the root of all evil.”

- Donald E. Knuth



Structured Programming with **go to** Statements

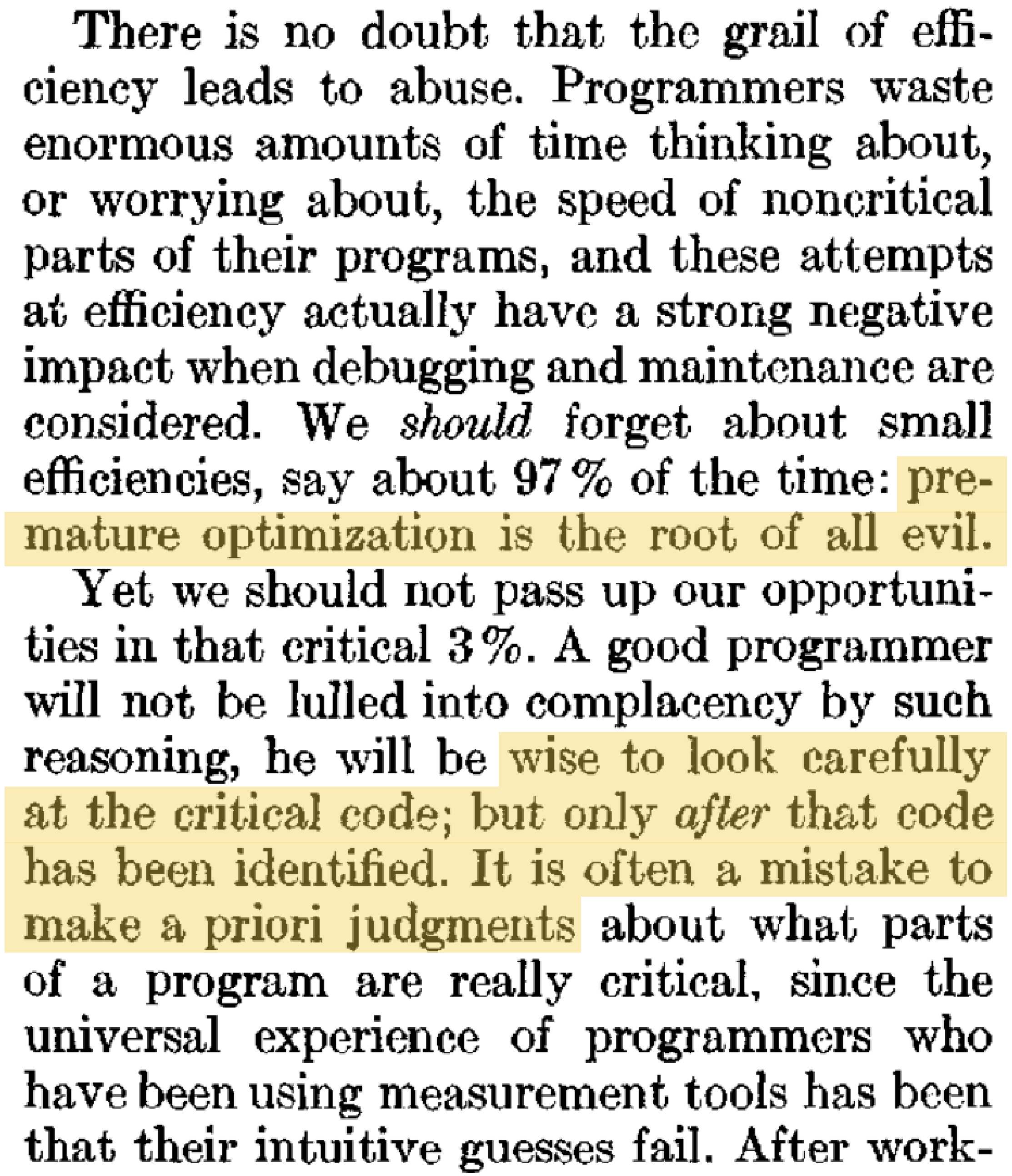
DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

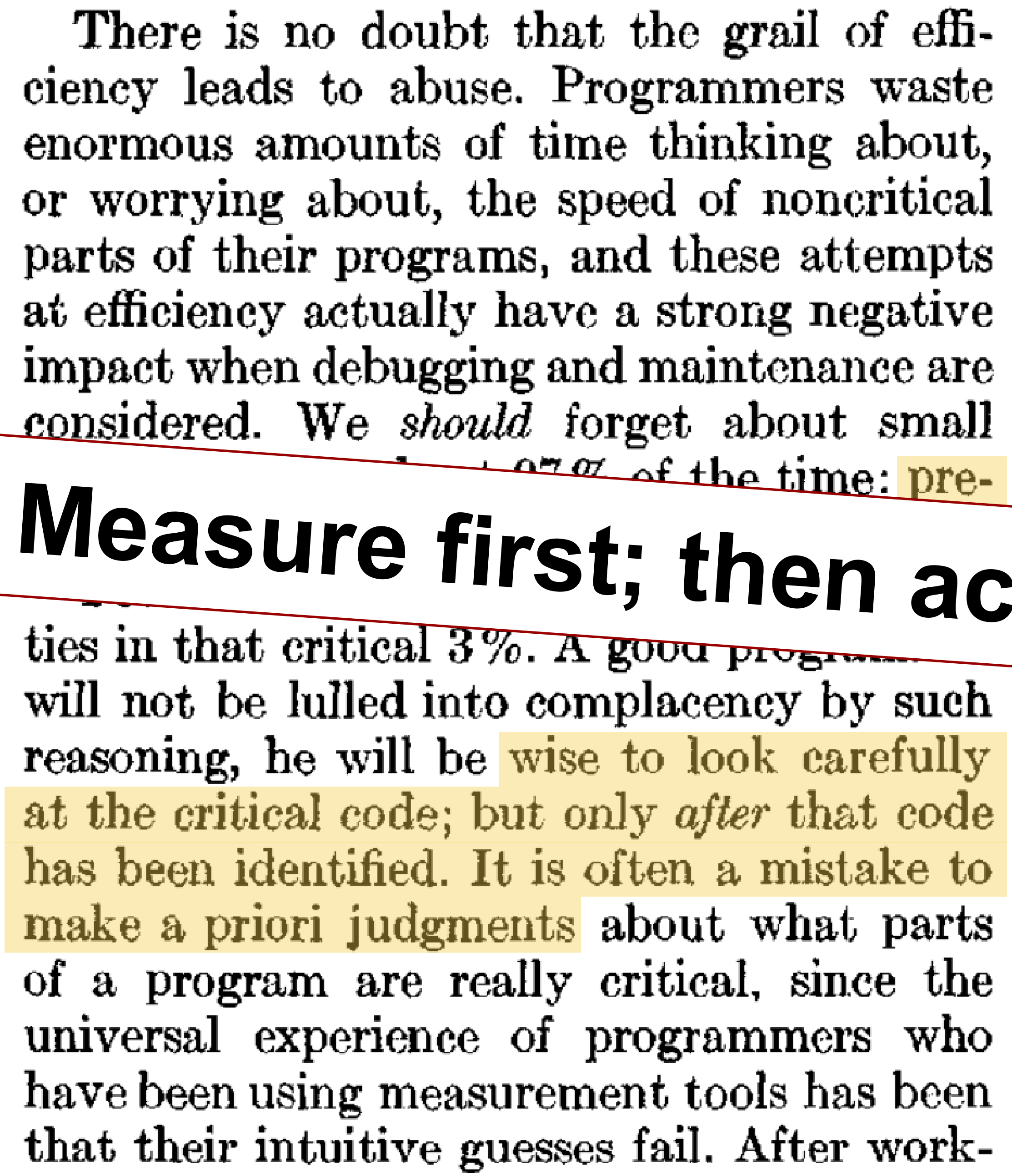
Keywords and phrases: structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

CR categories: 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)



There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. After work-


There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small

97% of the time: pre-

ties in that critical 3%. A good progra

will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. After work-

Measure first; then act

Profiling

So far…

```
t = time()
for _ in range(num_repetitions):

    2 * mat[:, 2]
t = time() - t
avg = t / num_repetitions
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python script.py
12
100
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
```

Run cProfile module

Profiling: function level

```
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
```

Sort results by cumulative time

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
8 function calls in 3.000 seconds

   Ordered by: cumulative time
                 tottime/ncalls     cumtime/ncalls
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.000    3.000 {built-in method builtins.exec}
        1    0.000    0.000    3.000    3.000 script.py:1(<module>)
        1    0.000    0.000    3.000    3.000 script.py:8(slow)
        1    3.000    3.000    3.000    3.000 {built-in method time.sleep}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of...
        1    0.000    0.000    0.000    0.000 script.py:3(fast)
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -s cumulative script.py
12
100
```

Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m cProfile -o script.prof script.py
12
100
```

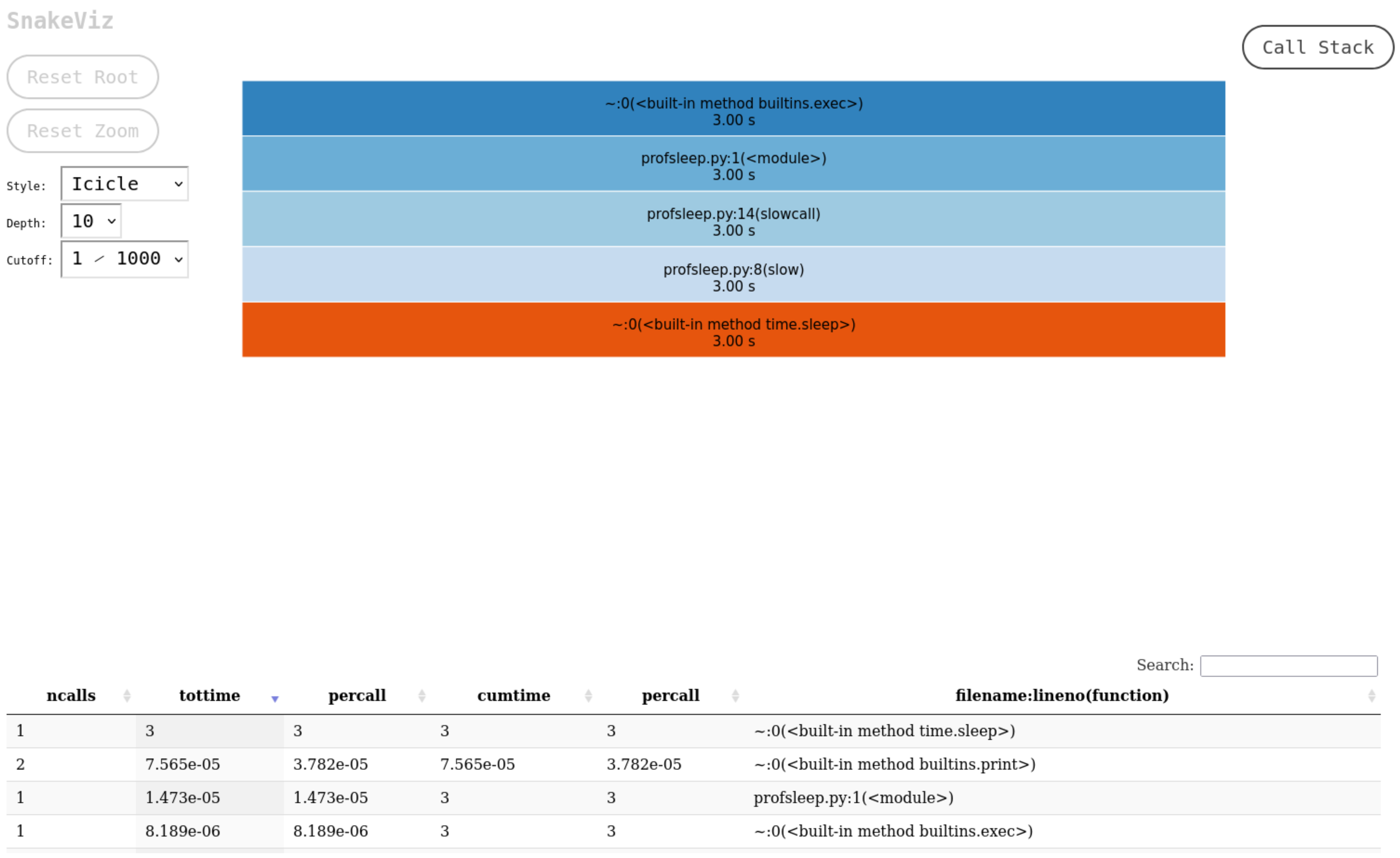
Profiling: function level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

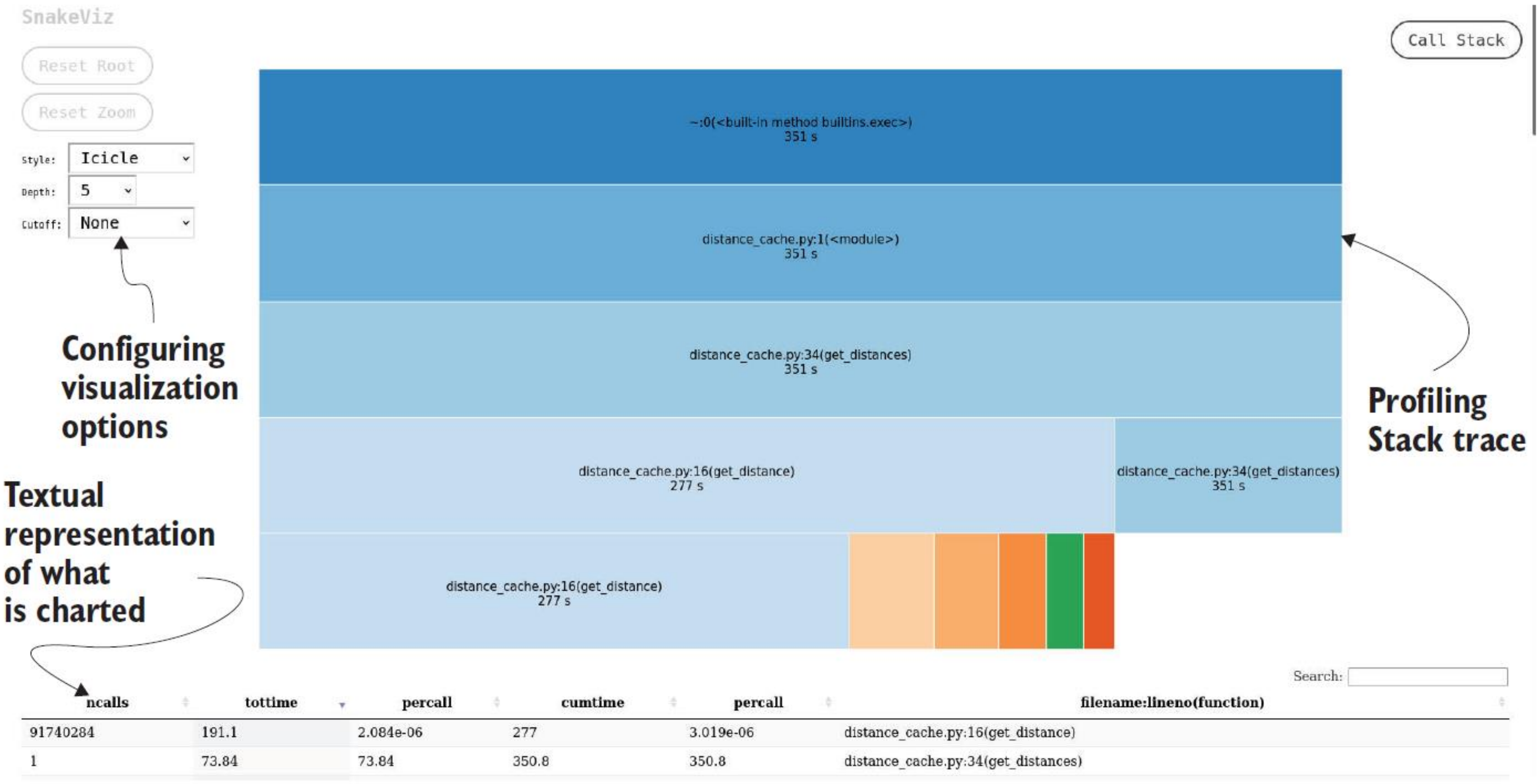
```
$ python -m cProfile -o script.prof script.py
12
100
$ snakeviz script.prof
```

Profiling: function level



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	3	3	3	3	~:0(<built-in method time.sleep>)
2	7.565e-05	3.782e-05	7.565e-05	3.782e-05	~:0(<built-in method builtins.print>)
1	1.473e-05	1.473e-05	3	3	profsleep.py:1(<module>)
1	8.189e-06	8.189e-06	3	3	~:0(<built-in method builtins.exec>)

Profiling: function level


SnakeViz
Reset Root
Reset Zoom
Style: Icicle
Depth: 5
Cutoff: None
Call Stack
~:0(<built-in method builtins.exec>)
351 s
distance_cache.py:1(<module>)
351 s
distance_cache.py:34(get_distances)
351 s
distance_cache.py:16(get_distance)
277 s
distance_cache.py:34(get_distances)
351 s
distance_cache.py:16(get_distance)
277 s
Configuring visualization options
Profiling Stack trace
Textual representation of what is charted
Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
91740284	191.1	2.084e-06	277	3.019e-06	distance_cache.py:16(get_distance)
1	73.84	73.84	350.8	350.8	distance_cache.py:34(get_distances)

Fast Python – Figure 2.1

Profiling: line level

```python
import time

def fast(x):
    result = x + 2
    return result

def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

Profiling: line level

```
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ kernprof -l script.py
12
100
Wrote profile results to script.py.lprof
Inspect results with:
python -m line_profiler -rmt “script.py.lprof”
```

Profiling: line level

```python
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: line level

```python
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: line level

```
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: line level

```python
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: line level

```python
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: line level

```python
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: line level

```
import time

def fast(x):
    result = x + 2
    return result

@profile
def slow(x):
    # Job security
    time.sleep(3)
    result = x * x
    return result

print(fast(10))
print(slow(10))
```

script.py

```
$ python -m line_profiler -rmt “script.py.lprof”
Timer unit: 1e-06 s

Total time: 3.00014 s
File: script.py
Function: slow at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def slow(x):
     9                                               # Job security
    10         1    3000132.9    3e+06    100.0      time.sleep(3)
    11         1          2.2      2.2      0.0      result = x * x
    12         1          0.6      0.6      0.0      return result

  3.00 seconds - script.py:7 - slow
```

Profiling: when to use what

Timing

Low overhead, very manual, no locations

Good for measuring performance

Function profiling

Low-ish overhead, automatic, rough locations

Good for zooming in on problem areas

Line profiling

High overhead, mostly automatic, precise locations

Good for pinning the exact problem, but should not be the start

Mini-project

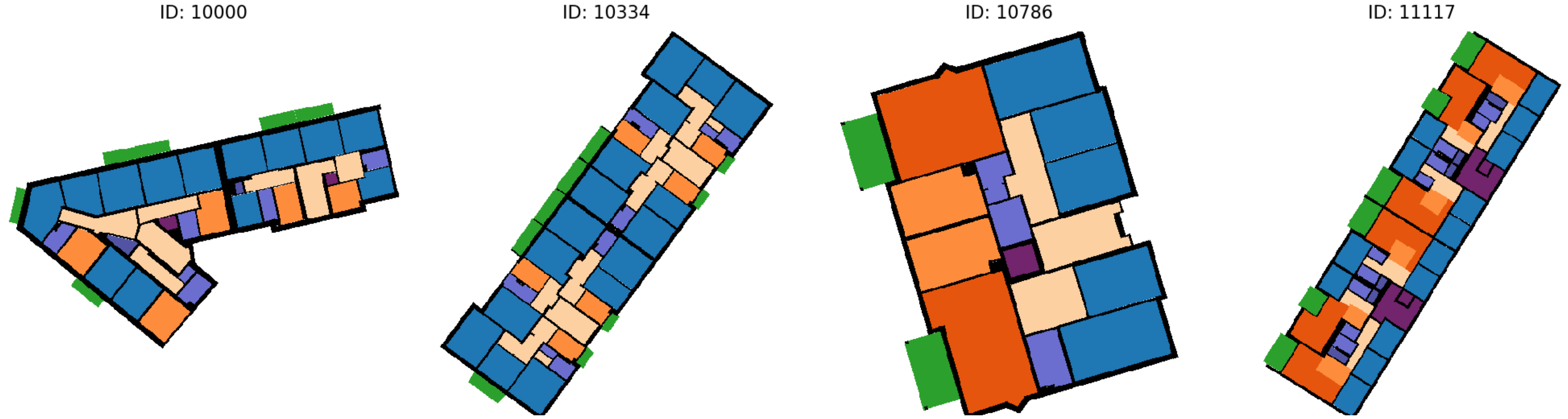
Mini-project now on Learn

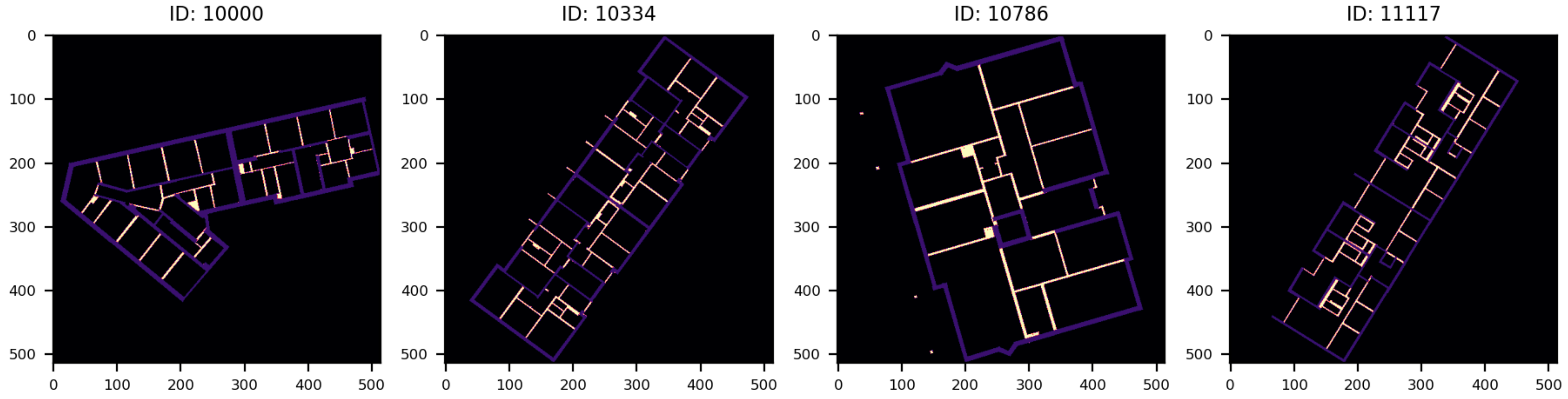
- Hand-in on Learn
  03.05.2026 (Sunday of semester week 12) 23:55

- Hand-in: PDF report and zip-file with code + job scripts in groups of 3-4. Make groups on Learn.

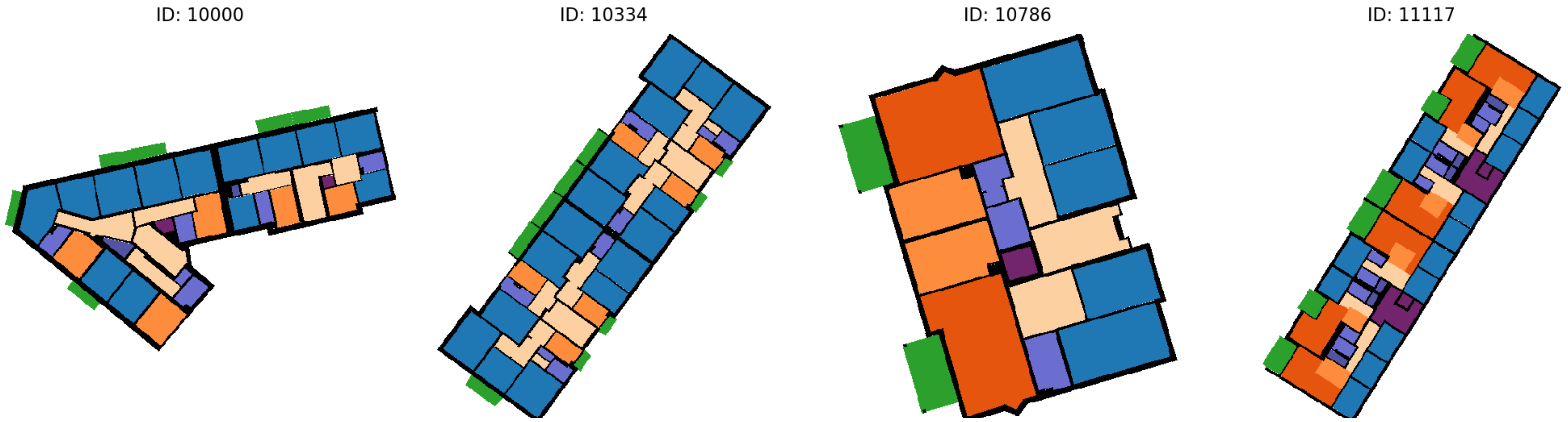
- Must *hand in and pass* project to attend the exam!

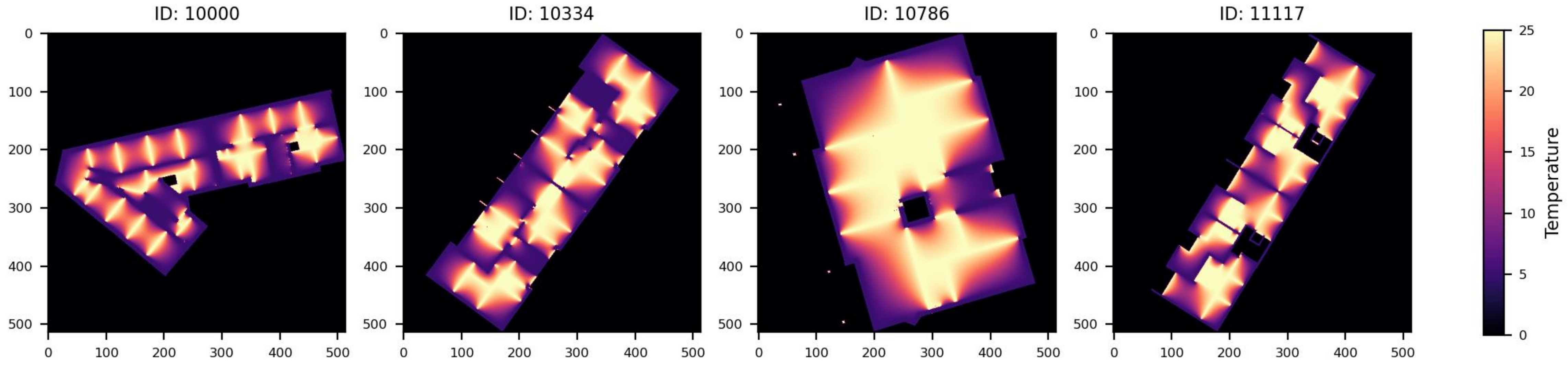
Mini-project: Wall Heating



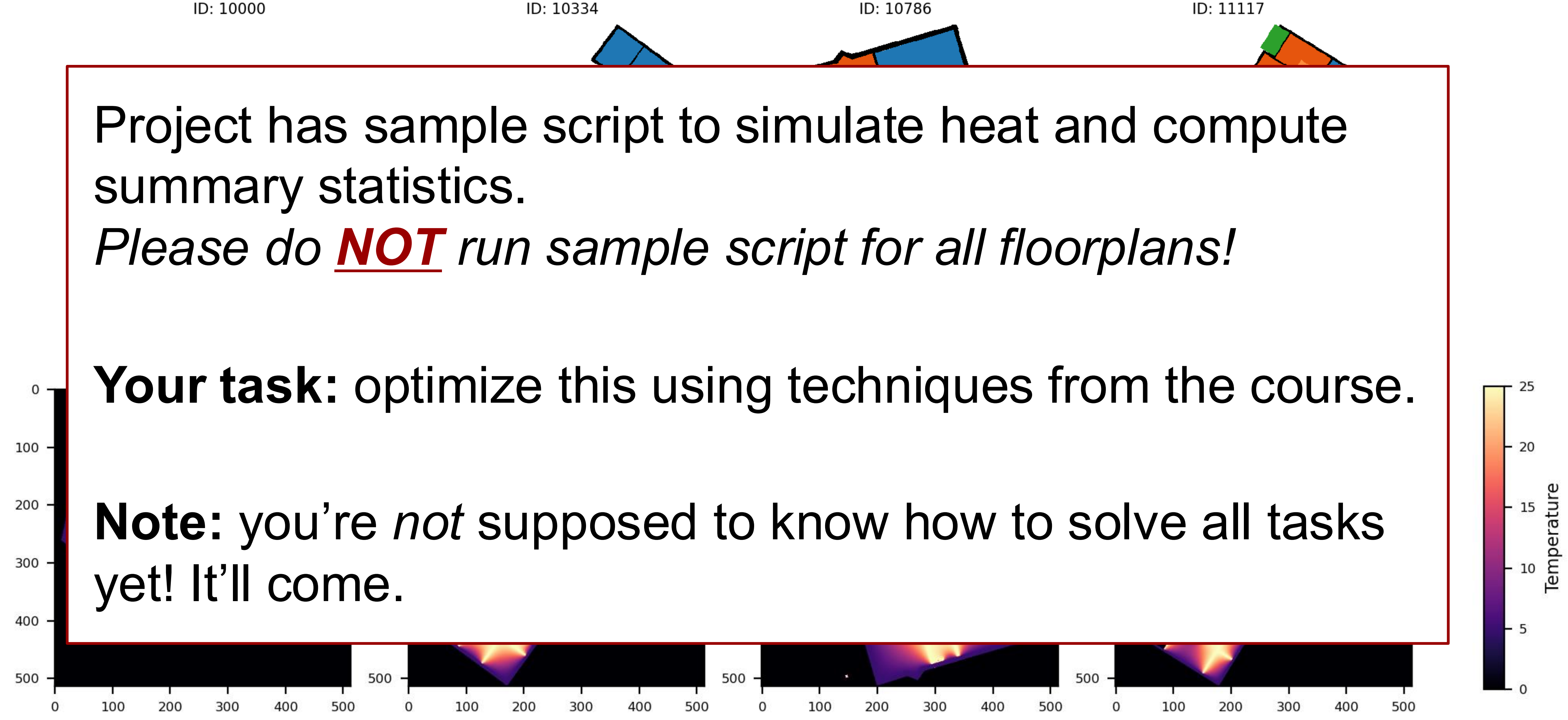


Mini-project: Wall Heating





Mini-project: Wall Heating



Project has sample script to simulate heat and compute summary statistics.
*Please do **NOT** run sample script for all floorplans!*

Your task: optimize this using techniques from the course.

Note: you're *not* supposed to know how to solve all tasks yet! It'll come.

Today’s exercise

Play with profiling and NumPy optimizations

- Use the different profilers to discover hotspots
- Fix them!
- Data is prepared on the cluster: /dtu/projects/02613_2025/data/locations/

DTU

Useful concepts

Add dimension to NumPy array

```
x[:, None]     # Add dimension at end
x[None]        # Add dimension at begin
x[:, None, :]  # Add dimension in the middle
```

cProfile for function profiling

```
python -m cProfile -s cumulative script.py  # In terminal

python –m cProfile –o script.prof script.py  # In snakeviz GUI
snakeviz script.prof
```

kernprof for line profiling

```
@profile  # Decorator for profiled functions

kernprof -l script.py                          # 1. Run profiling
python -m line_profiler -rmt “script.py.lprof”   # 2. Show results
```

Change to work node

```
linuxsh
```

Submit job script

```
bsub < submit.sh
```

Job status

```
bstat / bjobs
```

Check job output

```
bpeek / bpeek <JOBID>
```

Kill job

```
bkill <JOBID>
```