Introduction to coding with python™

Workshop 2 – 15-09-2023

Swinda Falkena – s.k.j.falkena@uu.nl – BBG 6.04
Institute of Marine and Atmospheric Research

Last workshop

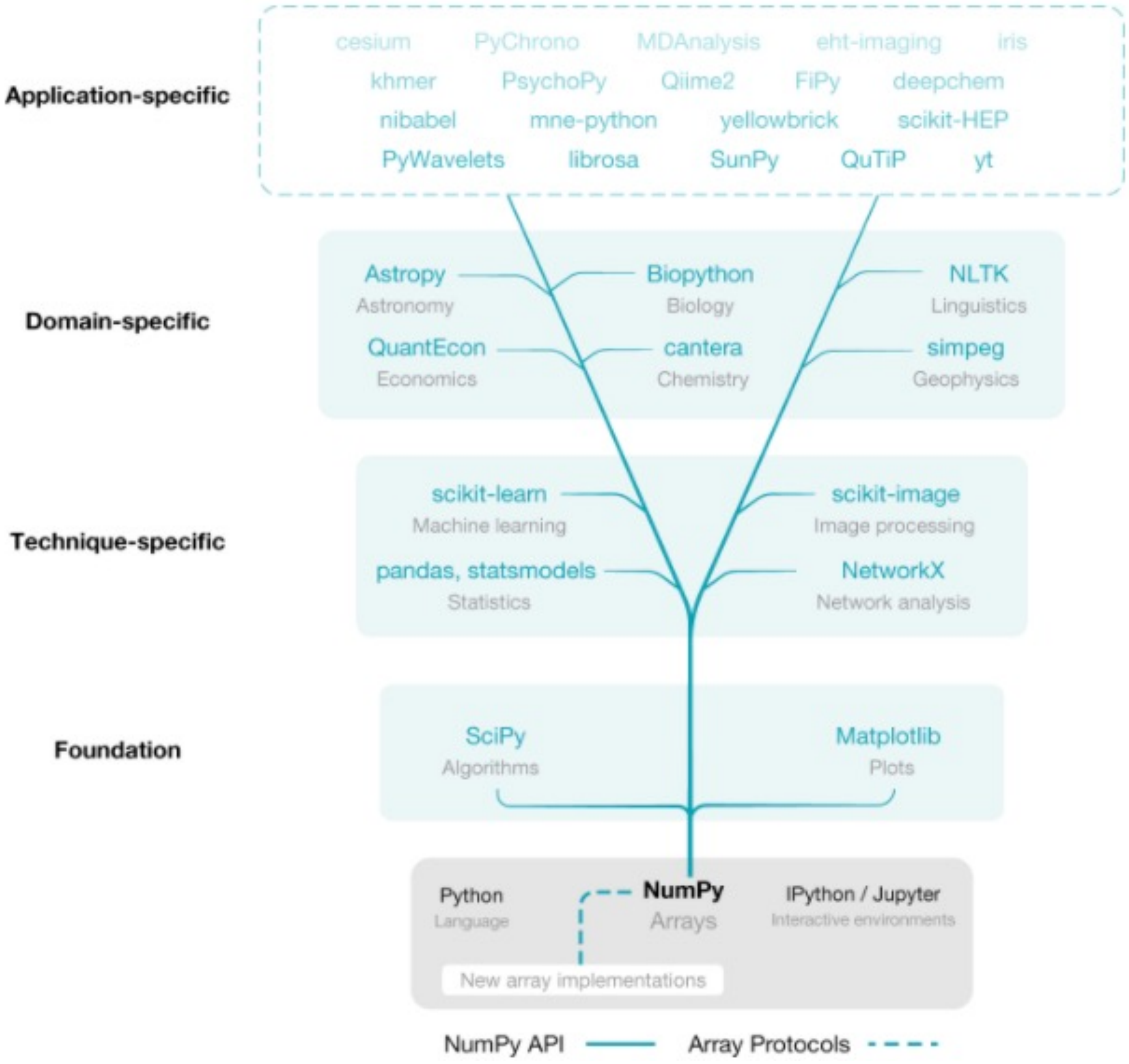
- What is Python and when do you use it?
- Creating folders, opening and saving a Notebook
- Cell types: code and markdown cells
- Using Python as a calculator and printing results
- Loops and creating functions
- Dealing with errors and commenting your code

Today

- Import modules: NumPy, Matplotlib and SciPy

- You can find the notebooks on Blackboard

Modules



Application-specific

cesium PyChrono MDAnalysis eht-imaging iris

khmer PsychoPy Qiime2 FiPy deepchem

nibabel mne-python yellowbrick scikit-HEP

PyWavelets librosa SunPy QuTiP yt

Domain-specific

Astropy
Astronomy

Biopython
Biology

NLTK
Linguistics

QuantEcon
Economics

cantera
Chemistry

simpeg
Geophysics

Technique-specific

scikit-learn
Machine learning

scikit-image
Image processing

pandas, statsmodels
Statistics

NetworkX
Network analysis

Foundation

SciPy
Algorithms

Matplotlib
Plots

Python
Language

NumPy
Arrays

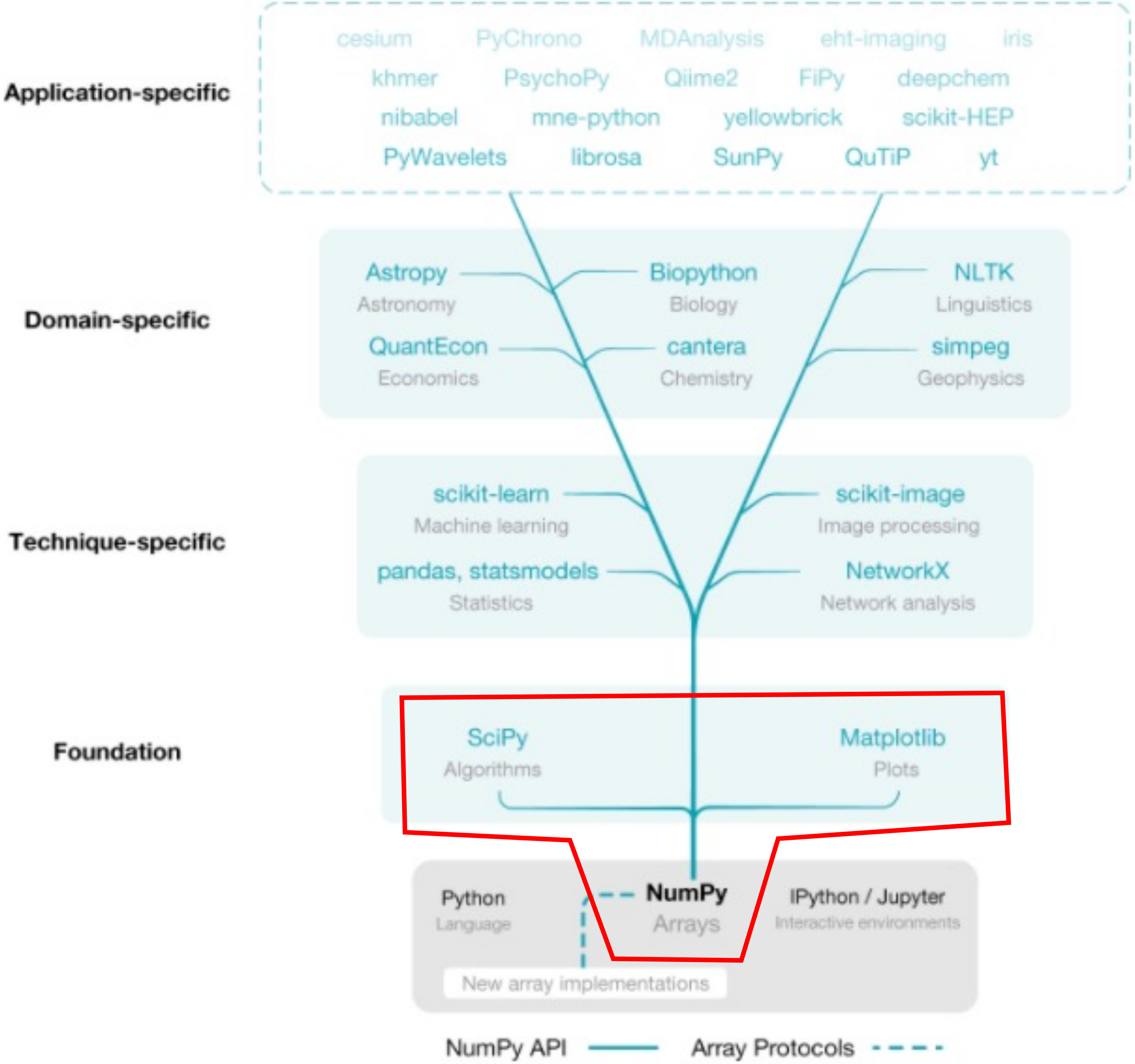
IPython / Jupyter
Interactive environments

New array implementations

NumPy API

Array Protocols

Modules


Application-specific
cesium
PyChrono
MDAnalysis
eht-imaging
iris
khmer
PsychoPy
Qiime2
FiPy
deepchem
nibabel
mne-python
yellowbrick
scikit-HEP
PyWavelets
librosa
SunPy
QuTiP
yt
Domain-specific
Astropy
Astronomy
Biopython
Biology
NLTK
Linguistics
QuantEcon
Economics
cantera
Chemistry
simpeg
Geophysics
Technique-specific
scikit-learn
Machine learning
scikit-image
Image processing
pandas, statsmodels
Statistics
NetworkX
Network analysis
Foundation
SciPy
Algorithms
Matplotlib
Plots
Python
Language
NumPy
Arrays
IPython / Jupyter
Interactive environments
New array implementations
NumPy API
Array Protocols

Importing Modules

- Most of useful functionalities of Python come from so-called packages or libraries (most already come with Anaconda).

- To use a library/package:

  1. import the package into your code

```
import matplotlib.pyplot as plt
import numpy as np
```

ALWAYS start your notebook with this!

Otherwise you have to type `matplotlib.pyplot` everytime you use it

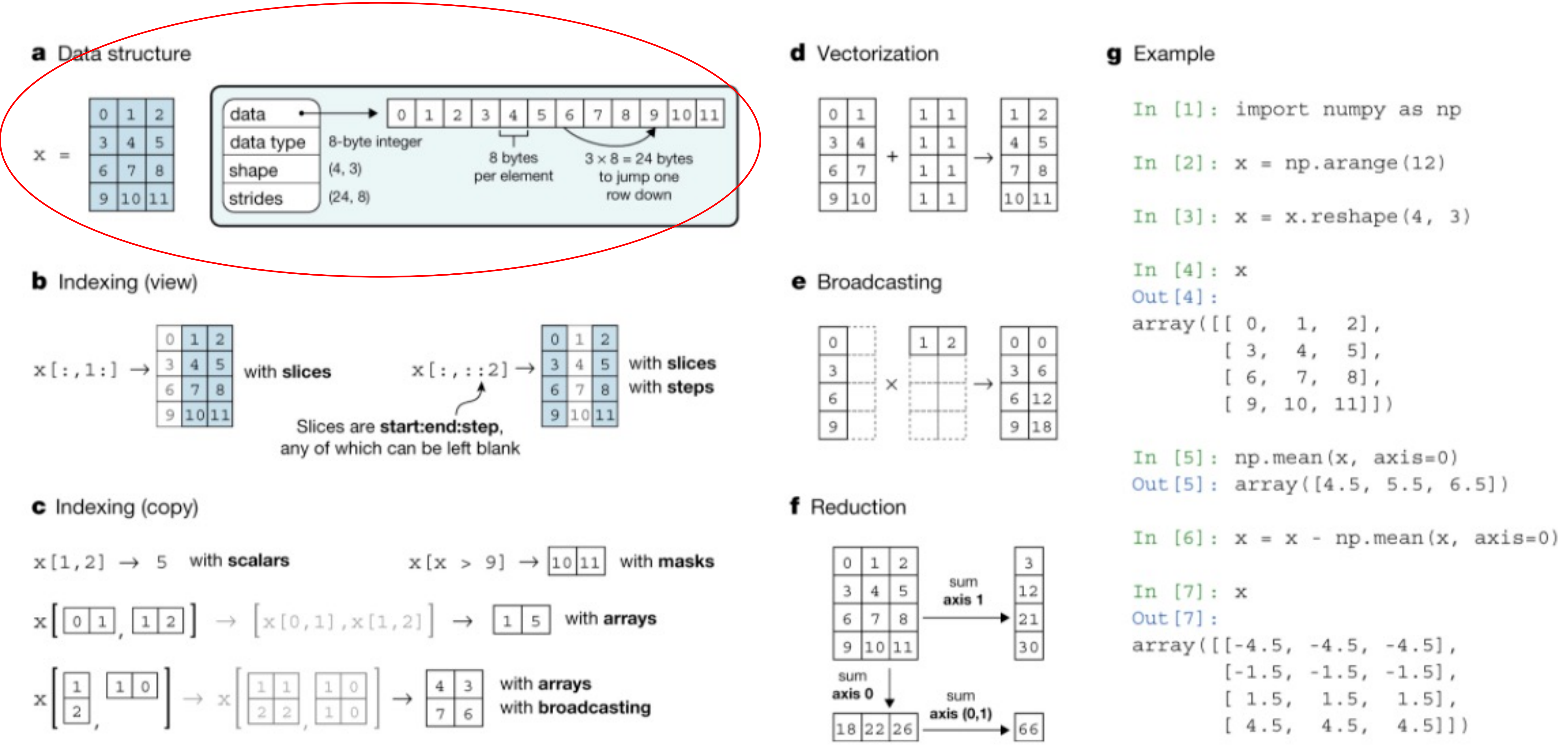
  2. use functions from the package by typing:

```
package.function_name

plt.function_name
np.function_name
```

NumPy

- Core is `ndarray` object: n-dimensional arrays of homogeneous data types

- All kinds of built-in operations for these data types
  → efficient way of dealing with large datasets



```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

Vectors and arrays

- `np.linspace(start,stop,number)`: creates a vector from `start` to `stop` of `number` linearly spaced numbers.

- `np.array([list])`: create a NumPy array from a list

- `np.arange(start, stop, step)`: creates a vector from `start` to `stop` with stepsize `step`.

Vectors and arrays

- `np.zeros(n)` = array full of zeros
- `np.ones(n)` = array full of ones
- `np.full(n,value)` = array of full with value `value`

Vectors and arrays

- `np.zeros(n)` = array full of zeros
- `np.ones(n)` = array full of ones
- `np.full(n,value)` = array of full with value `value`

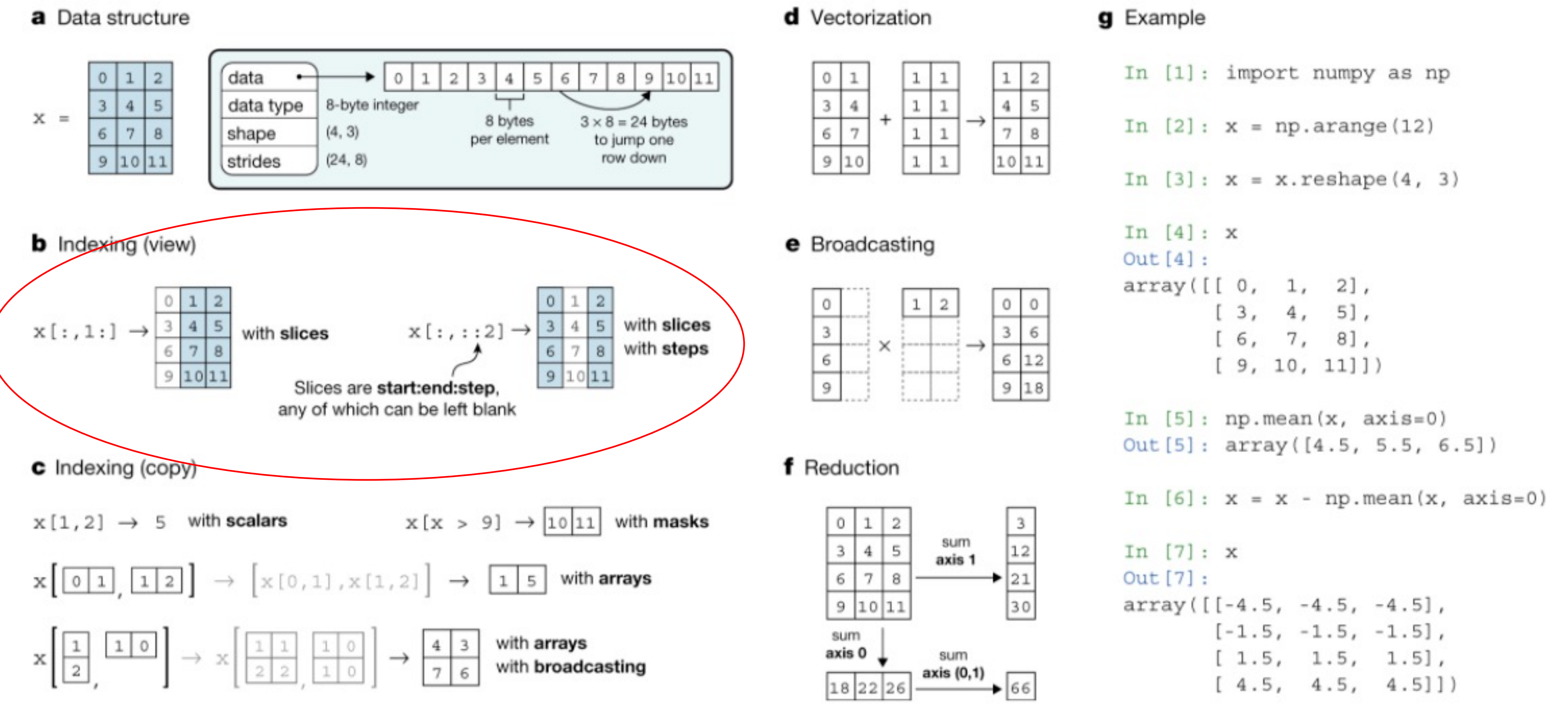
- `n` can be multidimensional: `c = np.zeros((9,9))`

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

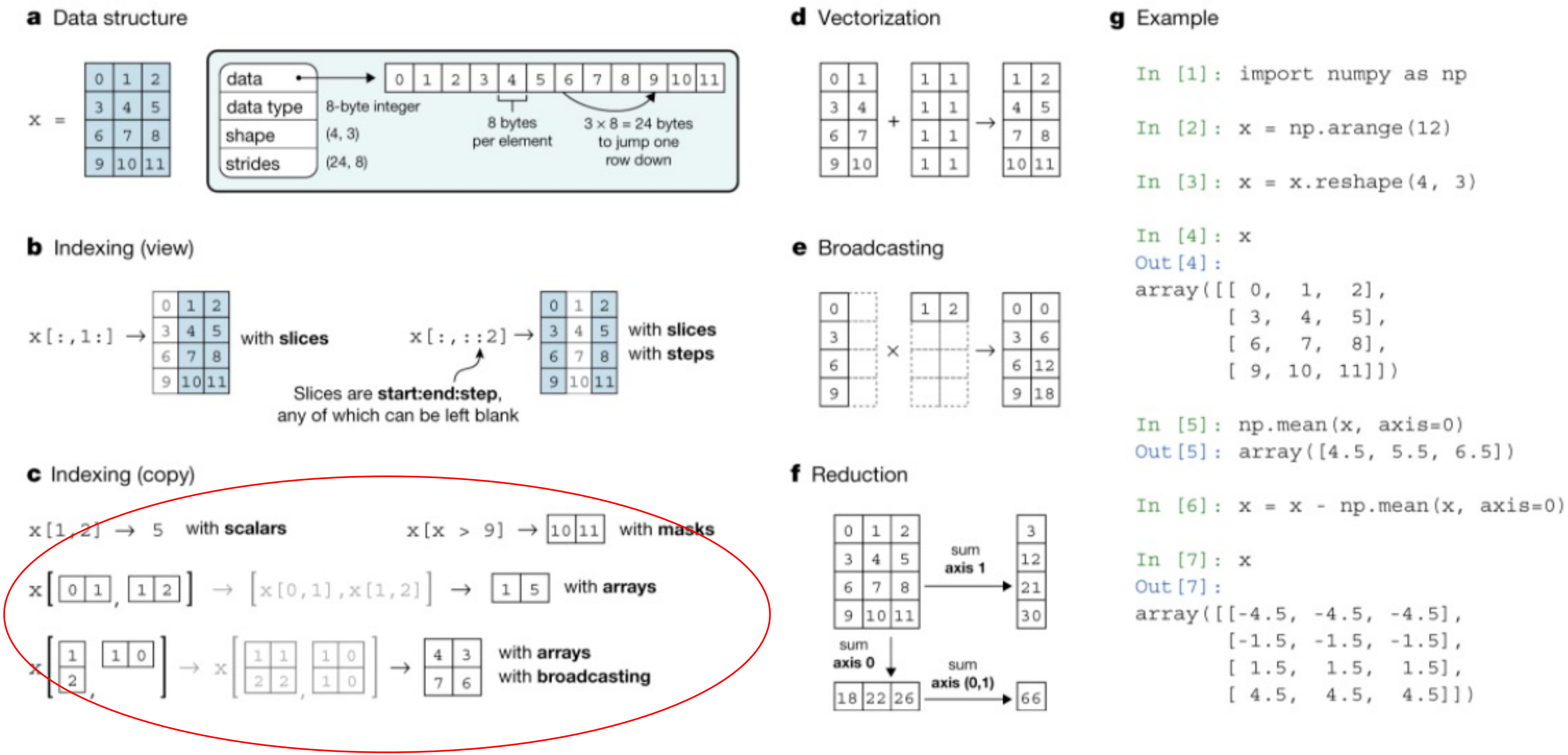
Vectors and arrays

- `np.zeros(n)` = array full of zeros
- `np.ones(n)` = array full of ones
- `np.full(n,value)` = array of full with value `value`

- `n` can be multidimensional: `c = np.zeros((9,9))`
- `c.shape = (9,9)`



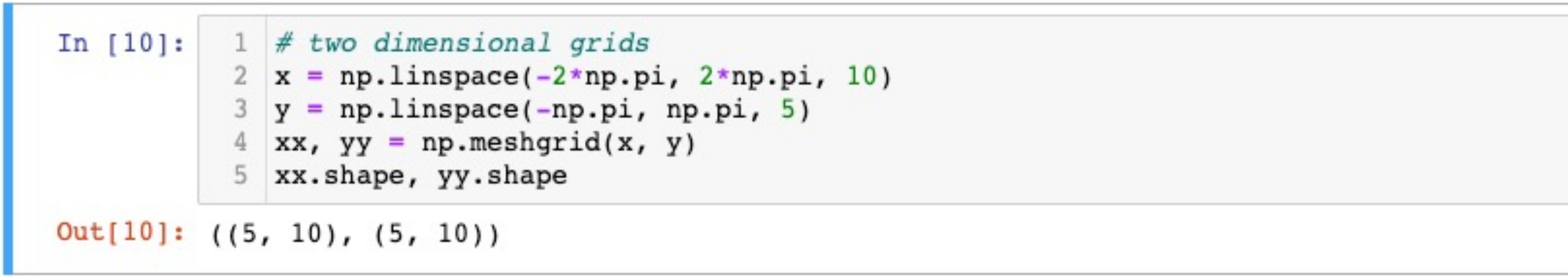
```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```



```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

Indexing

- Zero-based indexing!

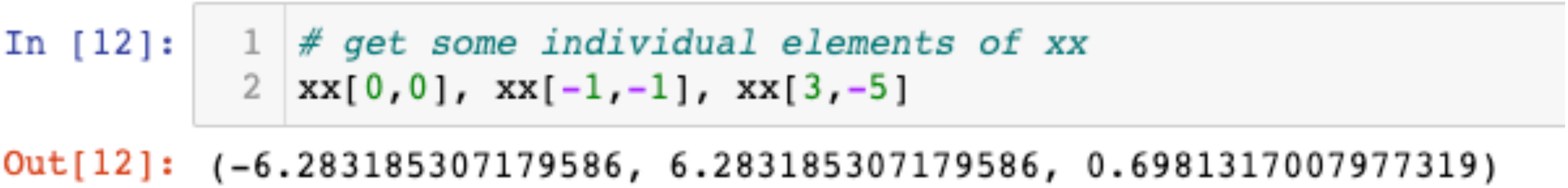


In [10]:

```
# two dimensional grids
x = np.linspace(-2*np.pi, 2*np.pi, 10)
y = np.linspace(-np.pi, np.pi, 5)
xx, yy = np.meshgrid(x, y)
xx.shape, yy.shape
```

Out[10]:

```
((5, 10), (5, 10))
```



In [12]:

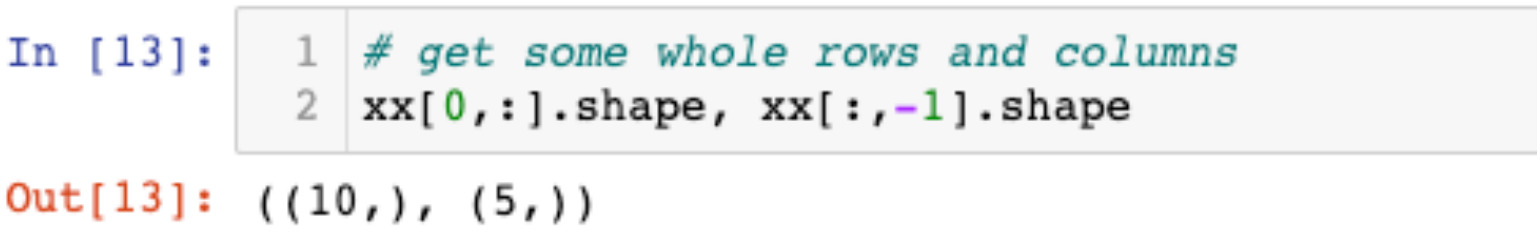
```
# get some individual elements of xx
xx[0,0], xx[-1,-1], xx[3,-5]
```

Out[12]:

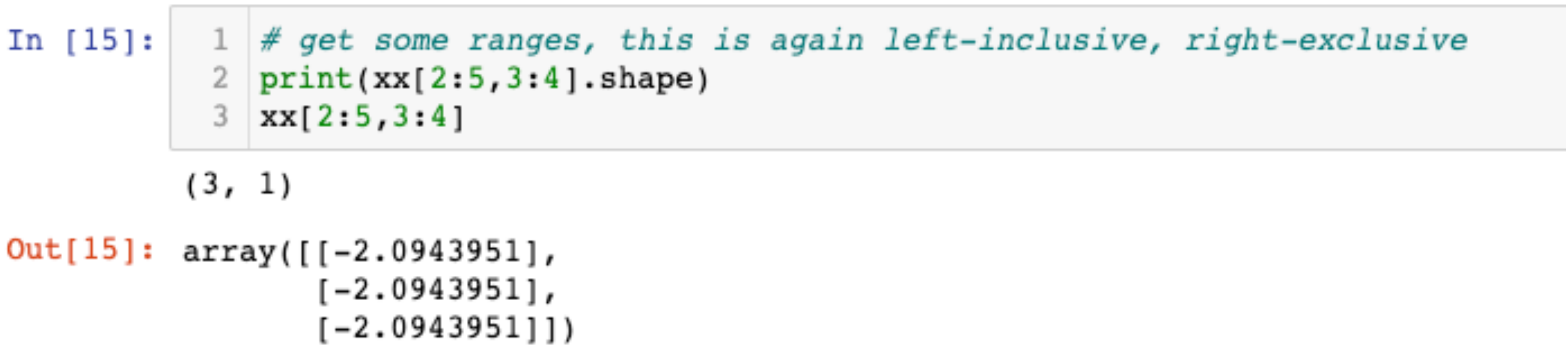
```
(-6.283185307179586, 6.283185307179586, 0.6981317007977319)
```

Indexing

- Zero-based indexing!



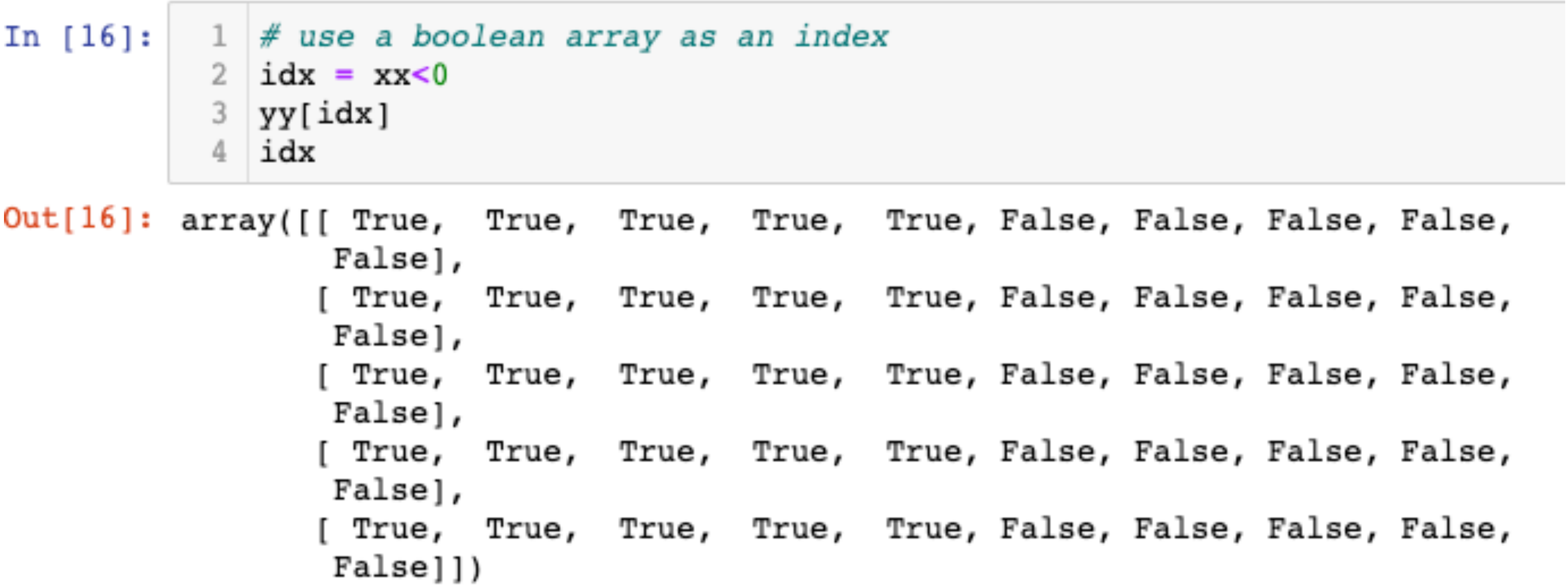
```
In [13]: # get some whole rows and columns
         xx[0,:].shape, xx[:,-1].shape

Out[13]: ((10,), (5,))
```



```
In [15]: # get some ranges, this is again left-inclusive, right-exclusive
         print(xx[2:5,3:4].shape)
         xx[2:5,3:4]

         (3, 1)

Out[15]: array([[-2.0943951],
                [-2.0943951],
                [-2.0943951]])
```

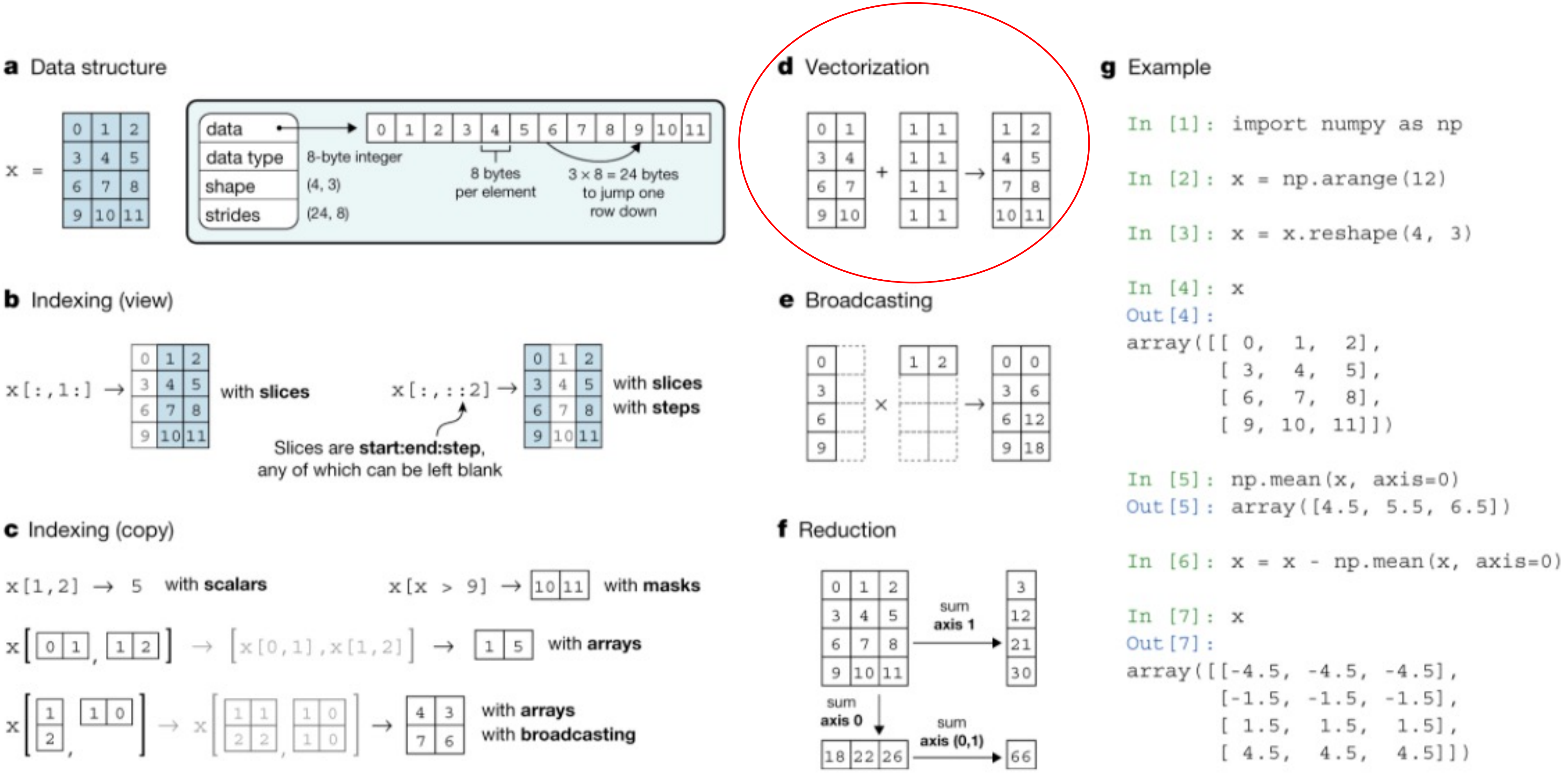
Indexing

- Zero-based indexing!



```
In [16]: # use a boolean array as an index
         idx = xx<0
         yy[idx]
         idx
```

```
Out[16]: array([[ True,  True,  True,  True,  True, False, False, False, False,
                 False],
                [ True,  True,  True,  True,  True, False, False, False, False,
                 False],
                [ True,  True,  True,  True,  True, False, False, False, False,
                 False],
                [ True,  True,  True,  True,  True, False, False, False, False,
                 False],
                [ True,  True,  True,  True,  True, False, False, False, False,
                 False]])
```



```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

Vectorization

- **Vectorization** (in Python context) = applying operations to whole arrays instead of individual elements

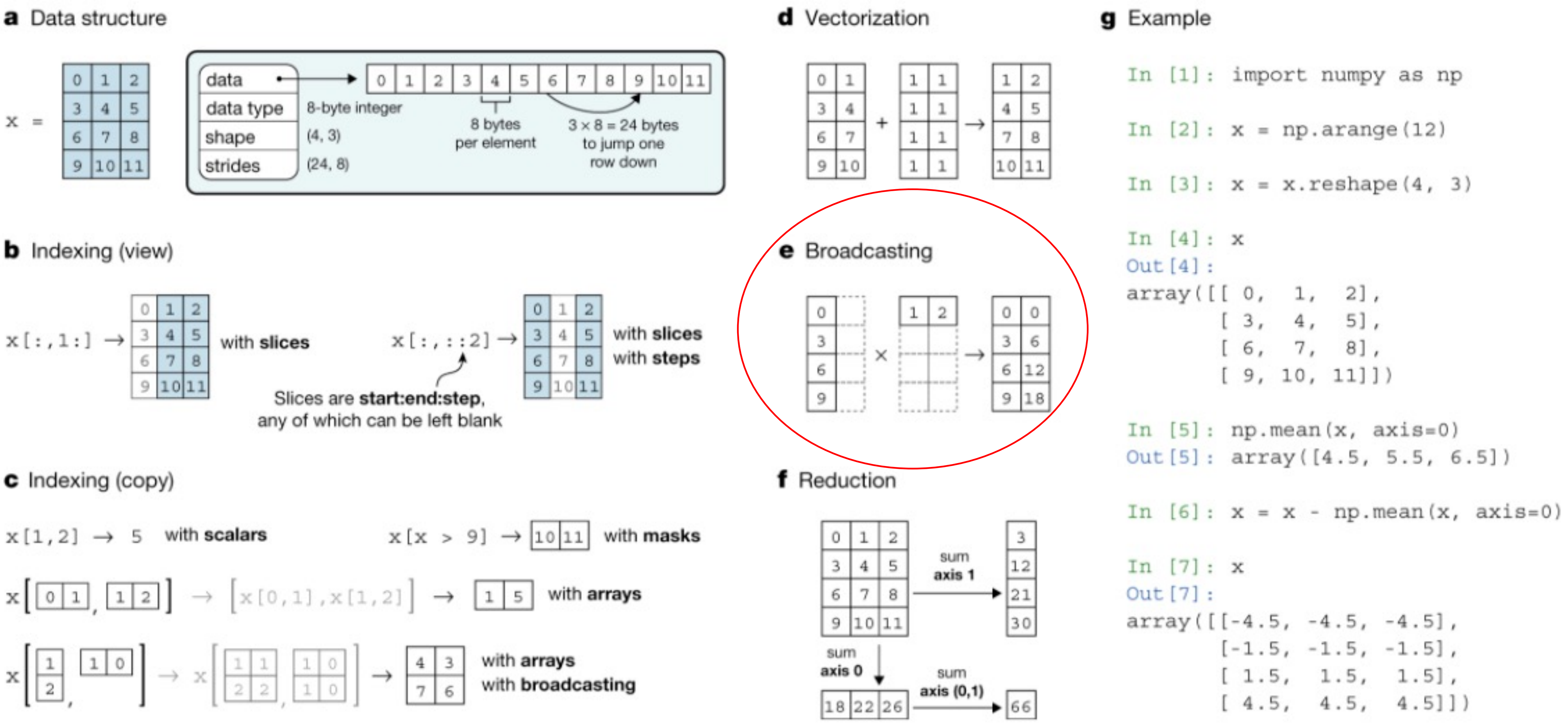
Vectorization

- **Vectorization** (in Python context) = applying operations to whole arrays instead of individual elements
- A lot more efficient than loops!

Vectorization

- **Vectorization** (in Python context) = applying operations to whole arrays instead of individual elements
- A lot more efficient than loops!

- `np.log(xx)`
- `np.sin(xx)`
- `np.cos(xx)`
- `np.exp(xx)`
- `np.pi`



```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

Broadcasting

- **Broadcasting** (in Python context) = operations on arrays with different shapes

Broadcasting

- **Broadcasting** (in Python context) = operations on arrays with different shapes
- Dimensions are compatible when:
  - they have the same length
  - one of them is 1

  → pay attention to the shape of your arrays!

Broadcasting

- **Broadcasting** (in Python context) = operations on arrays with different shapes
- Dimensions are compatible when:
    - they have the same length
    - one of them is 1

  → pay attention to the shape of your arrays!
- `F = np.zeros((5,10))`
- `X = np.linspace(0,2*np.pi,10)`

What are their shapes?

Broadcasting

- **Broadcasting** (in Python context) = operations on arrays with different shapes
- Dimensions are compatible when:

  - they have the same length
  - one of them is 1

  → pay attention to the shape of your arrays!
- `F = np.zeros((5,10))` → `(5,10)` 5 rows, 10 columns
- `X = np.linspace(0,2*np.pi,10)` → `(10,)` 10 rows, 1 column

Broadcasting

- **Broadcasting** (in Python context) = operations on arrays with different shapes
- Dimensions are compatible when:

  - they have the same length

  - one of them is 1

  → pay attention to the shape of your arrays!
- `F = np.zeros((5,10))` → `(5,10)`
- `X = np.linspace(0,2*np.pi,10)` → `(10,)`
- `D = F + X`

Broadcasting

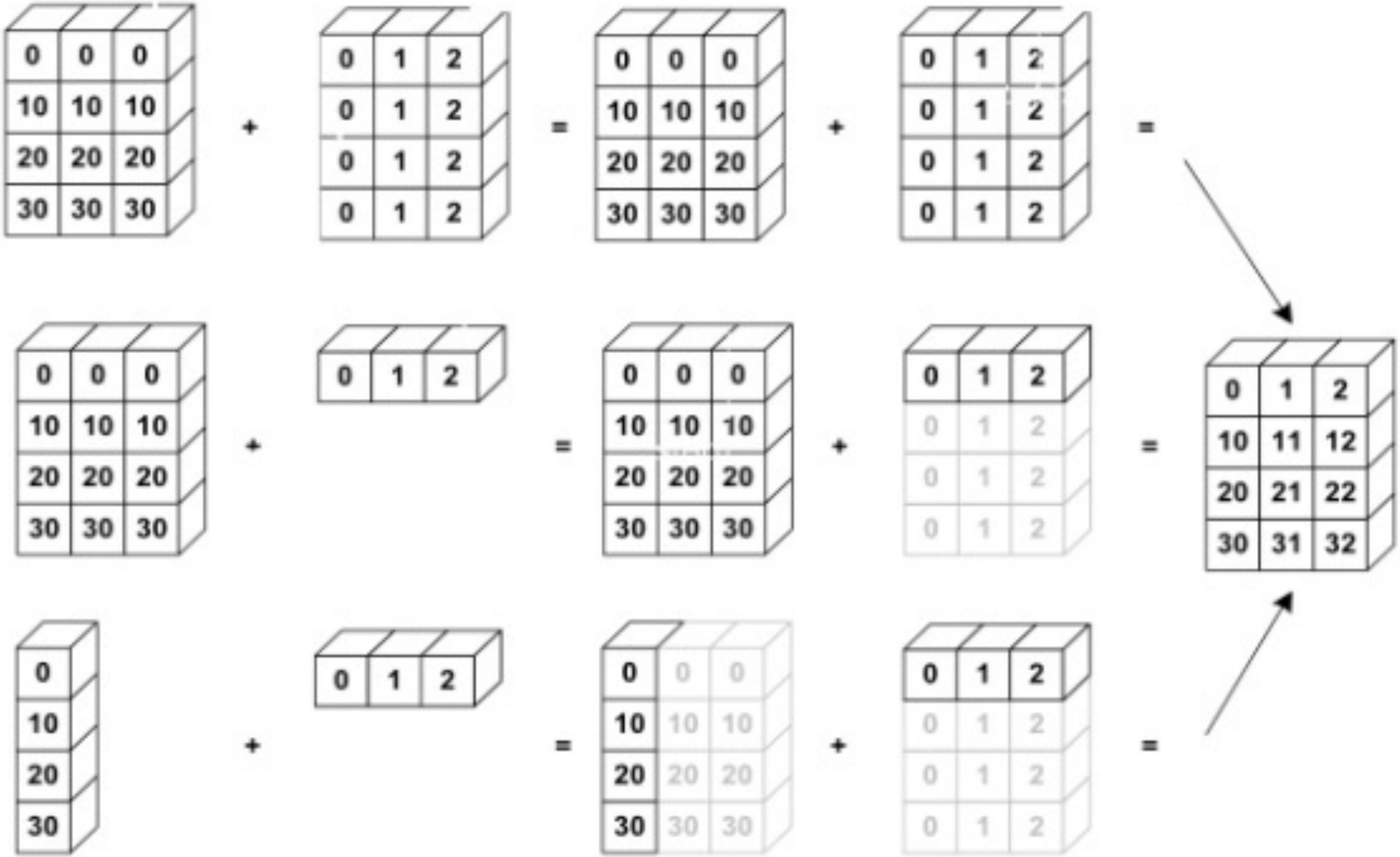
- **Broadcasting** (in Python context) = operations on arrays with different shapes
- Dimensions are compatible when:

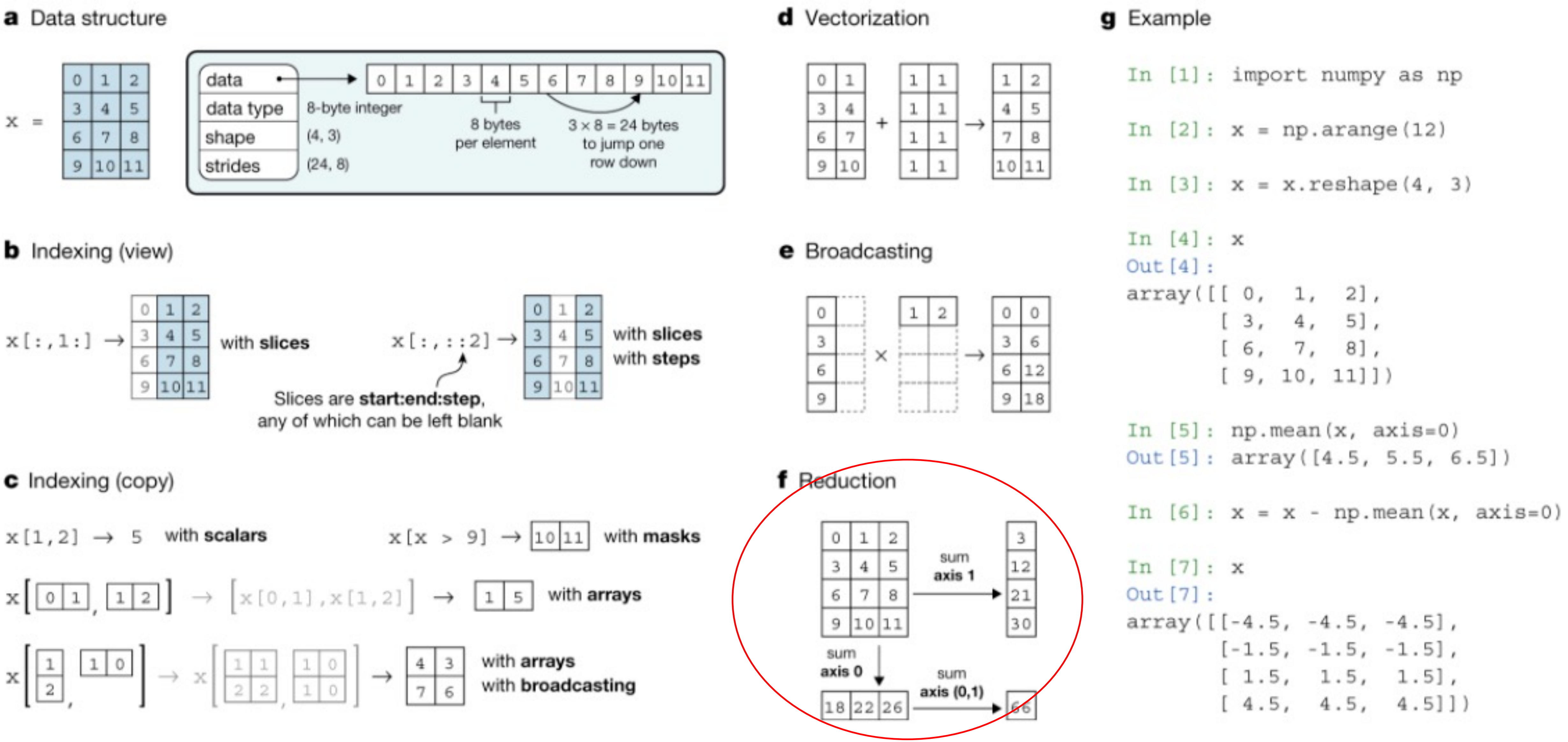
  - they have the same length

  - one of them is 1

  → pay attention to the shape of your arrays!
- `F = np.zeros((5,10))` → `(5,10)`
- `X = np.linspace(0,2*np.pi,10)` → `(10,)`
- `D = F + X`
- `G = F * X`

What if `X` had shape `(5,)`?


0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 1 2
10 11 12
20 21 22
30 31 32
0
10
20
30
+
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=



```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

Reduction

- **Reduction** (in Python context) = operations that collapse one or more dimension

Reduction

- **Reduction** (in Python context) = operations that collapse one or more dimension
- `X.sum()`
- `X.mean()`
- `X.std()`
- `X.max()`
- `X.min()`

Reduction

- **Reduction** (in Python context) = operations that collapse one or more dimension
- `X.sum()`
- `X.mean()`
- `X.std()`
- `X.max()`
- `X.min()`
- If an array has more than 1 dimension, you can also choose over which dimension to perform the computation

Matplotlib

- Library for visualizing and plotting data

Matplotlib

- Library for visualizing and plotting data
- Lineplots, scatterplots and contourplots

Plotting

Example of plotting a cosine wave:

- Combination of libraries `pyplot` and `numpy`
- Create an array *x* of 20 equally-spaced numbers between 0 and 2π:

```
x = np.linspace(0, 2*np.pi, 20)
```

- Use function plot:

```
plt.plot(x, np.cos(x))
plt.xlabel('x’)
plt.ylabel('y’)
plt.title('y = cos(x)’)
```

Plotting

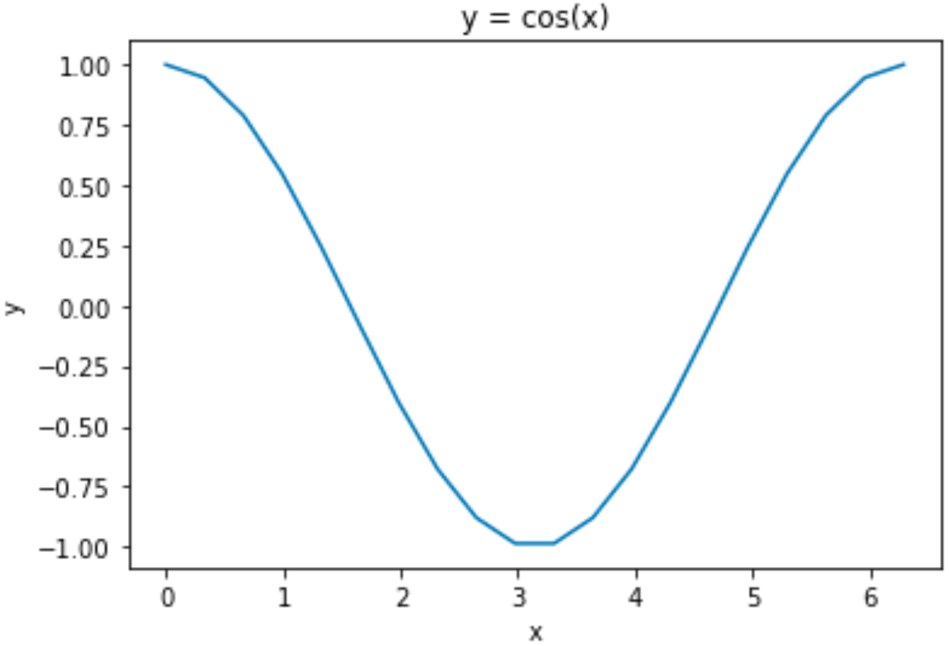
Example of plotting a cosine wave:

- Combination of libraries `pyplot` and `numpy`
- Create an array *x* of 20 equally-spaced numbers between 0 and 2π:

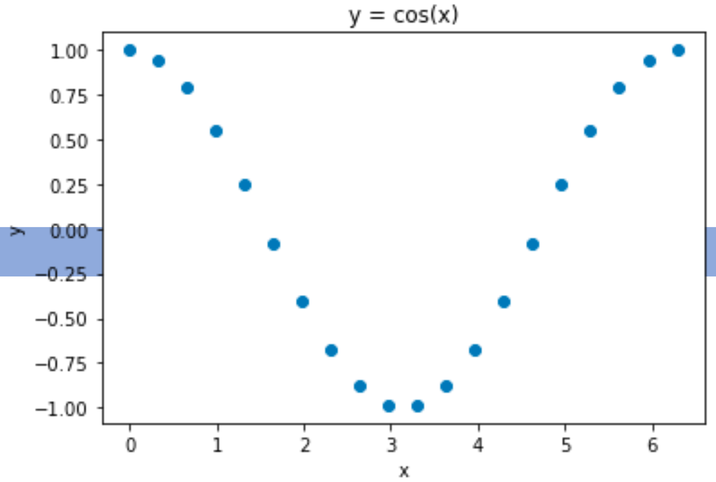
```
x = np.linspace(0, 2*np.pi, 20)
```

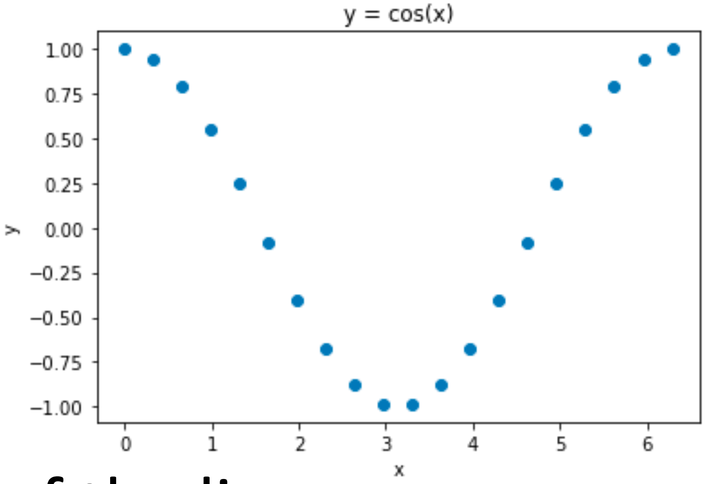
- Use function plot:

```
plt.plot(x, np.cos(x))
plt.xlabel('x’)
plt.ylabel('y’)
plt.title('y = cos(x)’)
```



Plotting





- The previous code plotted a solid line. We can also only plot the points (x, cos(x)) with the code below:

```
plt.plot(x, np.cos(x), 'o’)
plt.scatter(x,np.cos(x))
```

- By typing `help(plt.plot)`

you can obtain more information:

- how to change the colour or the linewidth of the lines
- how to prescribe the limits on the axes
- add a legend and title to the plot.

Contourplots

Often we want to plot **two-dimensional fields**

→ function `plt.contour()`

Contourplots

Often we want to plot **two-dimensional fields**
→ function `plt.contour()`

```
x = np.linspace(0, 10, 1000)
y = np.linspace(0, 10, 1000)
xx, yy = np.meshgrid(x,y)
```

Create an *x* and *y*-array, each has a length of 1000
Create a 2D grid from the arrays

Contourplots

Often we want to plot **two-dimensional fields**

→ function `plt.contour()`

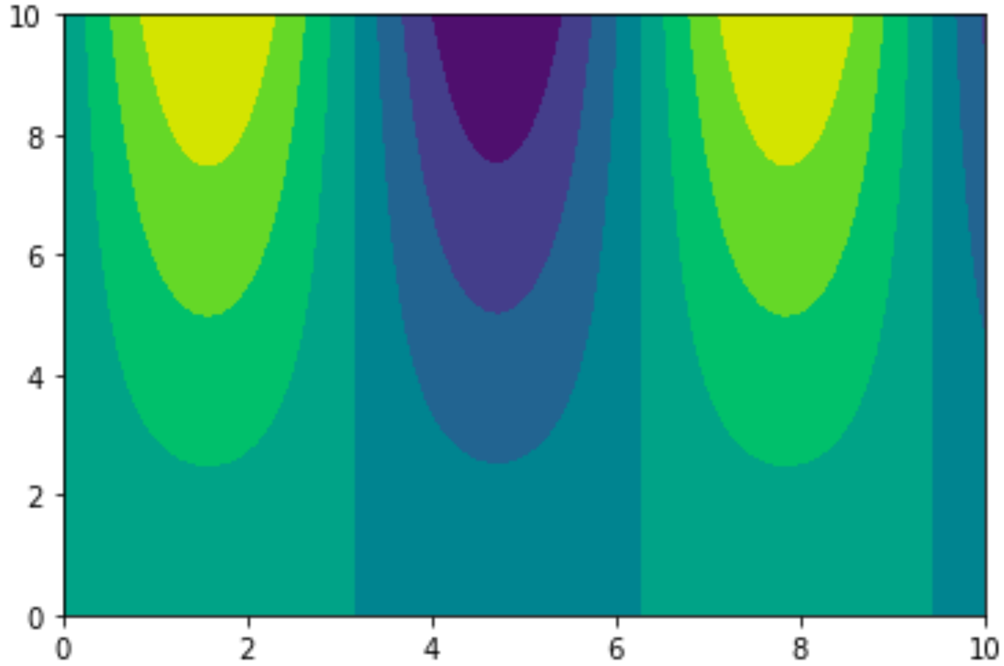
```
x = np.linspace(0, 10, 1000)
y = np.linspace(0, 10, 1000)
xx, yy = np.meshgrid(x,y)

z = np.sin(xx) * yy


plt.contourf(x,y,z)
```

Contourplots

Often we want to plot **two-dimensional fields**

→ function `plt.contour()`

```
x = np.linspace(0, 10, 1000)
y = np.linspace(0, 10, 1000)
xx, yy = np.meshgrid(x,y)

z = np.sin(xx) * yy


plt.contourf(x,y,z)
```



More plotting!

Interactive figures = figures than can be zoomed in and rotated

To achieve that, start you Notebook with:

```
%matplotlib notebook
```

If you do this, you have to tell Python each time a new figure starts, otherwise they will overlap.
So for each new figure, write:

```
plt.figure()
…
plt.show()
```

SciPy

SciPy = scientific computing package

SciPy

SciPy = scientific computing package

- Integrating
- Interpolating
- Curvefitting and optimizing
- Statistics
- Fourier transforms
- …

SciPy

SciPy = scientific computing package

- Integrating
- Interpolating
- Curvefitting and optimizing
- Statistics
- Fourier transforms
- …

Examples of interpolating and curvefitting in the (short) tutorial

Notebooks for today

- Workshop 2a – NumPy
- Workshop 2b – Matplotlib
- Workshop 2c – SciPy

On Blackboard (course content ACCP)!