MPS224 Scientific Computing

Dr. Sam Dolan

School of Mathematical and Physical Sciences,
University of Sheffield

Spring 2026

2. Working with arrays

G18 Hicks Building
s.dolan@sheffield.ac.uk

Today's lecture: Working with arrays

Examples:

- a one-dimensional array, **A**, of square numbers:

	A[0]	A[1]	A[2]	A[3]	…	A[n-1]
A	1	4	9	16		n^2

- a two-dimensional array, **B**, with elements *B*[*row, column*]:

B	0	1	2	…	m
0	B[0,0]	B[0,1]	B[0,2]	…	B[0,m]
1	B[1,0]	B[1,1]	B[1,2]	…	B[1,m]
2	B[2,0]	B[2,1]	B[2,2]	…	B[2,m]
…	…	…	…	…	…
n	B[n,0]	B[n,1]	B[n,2]	…	B[n,m]

NumPy

- NumPy is the fundamental package for scientific computing in Python.
- It enables us to work with arrays.
- It also has functions for linear algebra and matrices, as well as Fourier transforms etc.

Convention

Import the module as follows:

```
>>> import numpy as np
```

Today: Working with arrays in `numpy`

- How to create new arrays
- Working with multi-dimensional arrays
- Array indexing, slicing, and views.
- Universal functions, **vectorization** and **broadcasting**
- Linear algebra
- A challenge: the Mandelbrot set

Arrays

	A[0]	A[1]	A[2]	A[3]	...	A[n-1]
A	1	4	9	16		n^2

What is an array?

In essence, an array consists of:

- a block of memory – the raw data.
- an indexing scheme – how to locate elements.
- a data type – how to interpret the raw data.

Making a new array

There are many ways to make a new 1D array:

- `np.array([1,2,3])` : creating an array from a list
- `np.arange()` : evenly-spaced, specifying the step size
- `np.linspace()` : evenly-spaced, specifying the number of points
- `np.zeros()` : array set to zeros
- `np.ones()` : array set to 1s
- `np.empty()` : an uninitialized array

```
>>> np.array([1,2,3])
array([1, 2, 3])
>>> np.arange(0, 3, 1)
array([0, 1, 2])
>>> np.arange(0, 3, 0.5)
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5])
>>> np.linspace(0, 3, 5)
array([ 0.  ,  0.75,  1.5 ,  2.25,  3.  ])
>>> np.zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> np.ones(6)
array([ 1.,  1.,  1.,  1.,  1.,  1.])
>>> 2*np.ones(6)   # array of twos
array([ 2.,  2.,  2.,  2.,  2.,  2.])
>>> np.empty(2)
array([  6.91667204e-310,   6.91667204e-310])
```

Arrays

- Arrays are **homogeneous**: All elements in an array must be of the same **data type** (`dtype`).
- Each element is of a fixed size in memory.
- `numpy` supports a range of data types, including:

int8	A byte (-128 to 127)
uint8	An unsigned integer (0 to 255)
...	
int64	A 64-bit integer (-9223372036854775808 to 9223372036854775807)
float64	A double precision float: sign bit, 11 bits exponent, 52 bits mantissa.
complex128	A complex number, represented by two 64-bit floats: the real and imaginary components.

Arrays

- Arrays are **objects** with **attributes** and **methods**
- Example **attributes** of an array object `a`:
  - `a.dtype` : the data type of the array, e.g. `float64`.
  - `a.ndim` : the number of dimensions, e.g. `2`.
  - `a.shape` : a tuple with the shape of the array, e.g. `(3,2)`.
  - `...`
- Example **methods** of an array object `a`:
  - `a.sum()` : returns the sum of the elements
  - `a.mean()` : returns the mean average
  - `a.min()` : returns the minimum value
  - `a.sort()` : sorts the elements into ascending order
  - `...`

- Examples :

```
>>> a = np.linspace(0, 5, 11)
>>> b = np.arange(0, 11)
>>> print(a, b)
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5. ]
[ 0  1  2  3  4  5  6  7  8  9 10]
>>> a.dtype   # data type of array
dtype('float64')
>>> b.dtype
dtype('int64')
>>> a.shape
(11,)
>>> a.ndim  # number of dimensions
1
>>> a.size
11
```

Python syntax: tuples with one element

```
>>> (2,3)  # a tuple
(2, 3)
>>> 2,3  # If I omit the brackets I still get a tuple
(2, 3)
>>> # How do I write a tuple with just one element?
>>> (2) # This doesn't work ...
2
>>> (2,) # This is how to write a tuple with just one element
(2, )
```

Special values: `nan` and `inf`

- `nan` = not a number
- `inf` = infinity.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a / 0
array([ nan,  inf,  inf,  inf])
```

Arrays in `numpy`

`ndarray` : the N-dimensional array class in `numpy`

An `ndarray` is a multidimensional container of items of the **same type and size**.

The number of dimensions and items in an array is defined by its **shape**, which is a tuple of N positive integers that specify the sizes of each dimension.

The type of items in the array is specified by a **data-type object** (`dtype`), one of which is associated with each `ndarray` instance.

`http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html`

Making a new 2D array

- From a list-of-lists.
- By reshaping a 1D array.
- By **broadcasting** a pair of 1D arrays.

```
>>> np.array([[1,2], [3,4]])  # from a list-of-lists
array([[1, 2],
       [3, 4]])
>>> np.arange(4).reshape(2,2)  # by reshaping
array([[0, 1],
       [2, 3]])
>>> np.array([[1,2]]) * np.array([[3],[4]])  # by broadcasting
array([[3, 6],
       [4, 8]])
```

Re-shaping an array

```
>>> a = np.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a.reshape(3,4)  # 3 rows, 4 columns
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.reshape(2,2,3)   # a 3D array
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

Indexing and Slicing

- indexes start from zero
- we can use indices to access or modify a single element of the array:

```
>>> a = np.arange(6).reshape(2,3)  # create a 2x3 array
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> a[1,2]  # get element in the 2nd row, 3rd column
5
>>> a[1,2] = -1  # modify this element
>>> print(a)
[[ 0  1  2]
 [ 3  4 -1]]
```

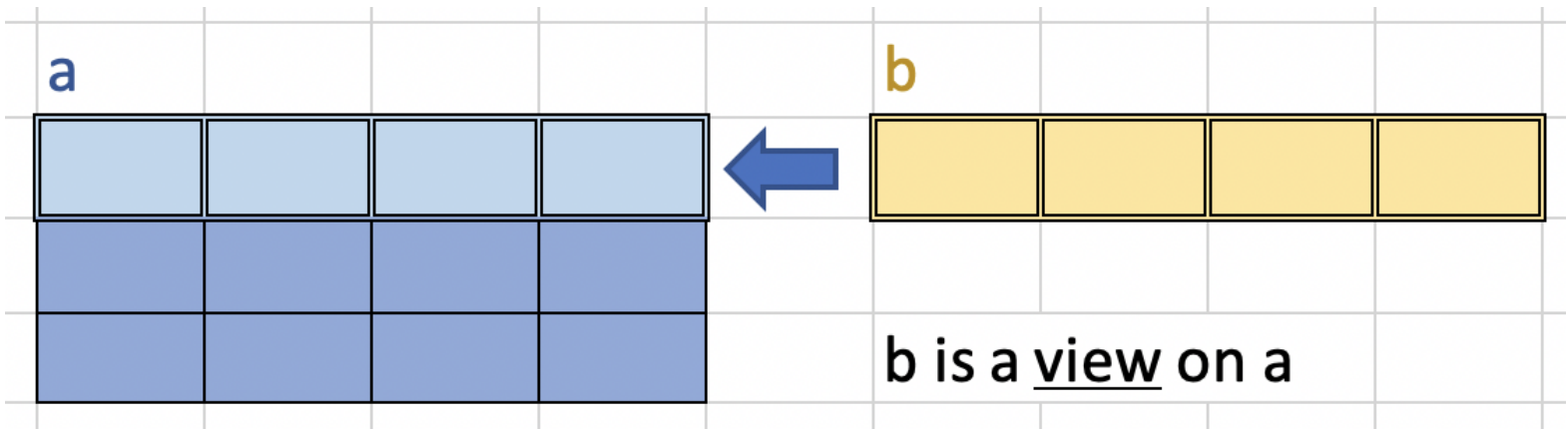
Indexing and Slicing

- Using **slicing**, we can access subarrays :

```
>>> a = np.arange(12).reshape(3,4)  # create a 3x4 array
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print(a[0,:]) # the 1st row
[0 1 2 3]
>>> print(a[:, 1]) # the 2nd column
[1 5 9]
>>> print(a[::-1, :])  # reverse the order of the rows
[[ 8  9 10 11]
 [ 4  5  6  7]
 [ 0  1  2  3]]
```

Views

- Slices of arrays are **not** new arrays.
- Slices are **views** on the original array.
- Example: `b = a[0,:]`



Views

- Example in code:

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a[0, :]  #  the first row
>>> b[2] = 99    #  By changing an element of b  ...
>>> print(a)     # ... we are changing the data in a
[[ 0  1 99  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Universal functions (ufuncs)

- A `ufunc` is a function that operates on **whole arrays** in an element-by-element fashion.
- Examples:

```
>>> a = np.linspace(0, 2*np.pi, 7)
>>> np.sin(a)  # sin is an example of a ufunc
array([  0.00000000e+00,   8.66025404e-01,   8.66025404e-01,
         1.22464680e-16,  -8.66025404e-01,  -8.66025404e-01,
        -2.44929360e-16])
>>> # multiplication, addition and raising-to-the power are ufuncs
>>> 3*a + a**2
array([  0.        ,   4.23821536,  10.66967615,  19.29438236,
        30.11233399,  43.12353105,  58.32797353])
```

Universal functions (ufuncs)

- Standard `ufuncs` are typically implemented in compiled C code $\Rightarrow$ they are **fast**
- Example: multiplying by 2 two different ways

```
>>> # Using list comprehension:
>>> a = range(1000)
>>> %timeit [2*n for n in a]
10000 loops, best of 3: 65.3 microsec per loop

>>> # Using an array with multiplication ufunc
>>> a = np.arange(1000)
>>> %timeit 2*a
1000000 loops, best of 3: 1.59 microsec per loop
```

- i.e. the latter is approximately 40 times faster

Vectorization

- **Vectorization** is the practice of replacing `while` and `for` loops with whole-array expressions and `ufuncs` ...
- ... so that batch operations are implemented efficiently.

Vectorization: Exercise

Exercise

- Generate 1000 random numbers from the standard uniform distribution $[0, 1)$
- Compute the variance of the sample using:
  1. lists and `for` loops
  2. arrays and `ufuncs`
- Compare the efficiencies of the two implementations.

$$\mathrm{var}(x) = \left\langle x^2 \right\rangle - \langle x \rangle^2$$

Implementation #1 (with loops)

```
import random as rnd
def attempt1(n=1000):
    l = []
    for i in range(n):    # Build a list of random numbers
        l.append(rnd.random())
    xsum = 0; x2sum = 0;
    for i in range(n):
        xsum += l[i]
        x2sum += l[i]**2
    xmean = xsum / n
    x2mean = x2sum / n
    return x2mean - xmean**2
```

Implementation #2 (vectorized)

- Use the `random` module within the `numpy` package to create *whole arrays* of random numbers.
- Use only `ufunc`'s on the arrays.

```
import numpy as np
def attempt2(n=1000):
    r = np.random.random(n)
    xmean = r.sum() / n
    x2mean = (r**2).sum() / n
    return x2mean - xmean**2
```

Comparing efficiencies

```
>>> %timeit attempt1()
1000 loops, best of 3: 372 us per loop

>>> %timeit attempt2()
100000 loops, best of 3: 17.8 us per loop
```

- The vectorized version is $\sim$ 20 times faster.

Broadcasting

What is broadcasting?

- An efficient way of combining two arrays of different shapes during arithmetic operations.
- The smaller array is ‘broadcast’ across the larger array
- The loops are carried out in C rather than Python
  $\Rightarrow$ **fast!**

Broadcasting

Example: Arrays of same size

- If the arrays are the same size, they are combined element-by-element:

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> a * b
array([ 4, 10, 18])
>>> a + b
array([5, 7, 9])
>>> a - b
array([-3, -3, -3])
>>> a**b
array([  1,  32, 729])
```

Broadcasting

Example: Scalar $\times$ vector

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

Broadcasting

A more interesting example

```
>>> x = np.arange(4)
>>> x2 = x.reshape(4, 1)
>>> y = np.arange(5)
>>> print(x.shape, x2.shape, y.shape)
(4,) (4, 1) (5,)
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,)
>>> x2 + y
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

Broadcasting

Broadcasting rules for combining two arrays:

- Their shapes are compared, starting with last dimension
- Two dimensions are compatible iff:
  - They are the same, or
  - One or both of them is 1.
- If dimensions are not compatible, an error is thrown
- **Note:** the arrays do not need to have same number of dimensions
- Example:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
A*B    (4d array):  8 x 7 x 6 x 5
```

- More info: `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`

Broadcasting

Warning!

Broadcasting is **not** the same as matrix multiplication!
For example,

```
>>> A = np.arange(4).reshape(2,2)
>>> b = np.arange(1,3).reshape(2,1)
>>> print(A)
[[0 1]
 [2 3]]
>>> print(b)
[[1]
 [2]]
>>> A * b   # Warning : not matrix multiplication
array([[0, 1],
       [4, 6]])
```

Broadcasting

Matrix multiplication

For matrix multiplication, use @ or `np.dot()` or first convert arrays to matrices :

```
>>> A @ b
array([[2],
       [8]])

>>> np.dot(A, b)
array([[2],
       [8]])

>>> np.mat(A) * np.mat(b)
matrix([[2],
        [8]])
```

Linear algebra

- Matrix & vector multiplication: use @ or the `np.dot` function
- The scalar product $\mathbf{a} \cdot \mathbf{b}$:

```
>>> a = np.array([1, 2])
>>> b = np.array([3, 4])
>>> a @ b   # scalar product
11

>>> np.dot(a,b)   # scalar product
11
```

Linear algebra

- Matrix multiplication:

```
>>> A = np.array([[1,2], [3,4]])
>>> b = np.array([5,6])
>>> A @ b   # matrix-by-vector
array([17, 39])

>>> A @ A  # matrix-by-matrix
array([[ 7, 10],
       [15, 22]])
```

http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Linear algebra with `numpy.linalg`

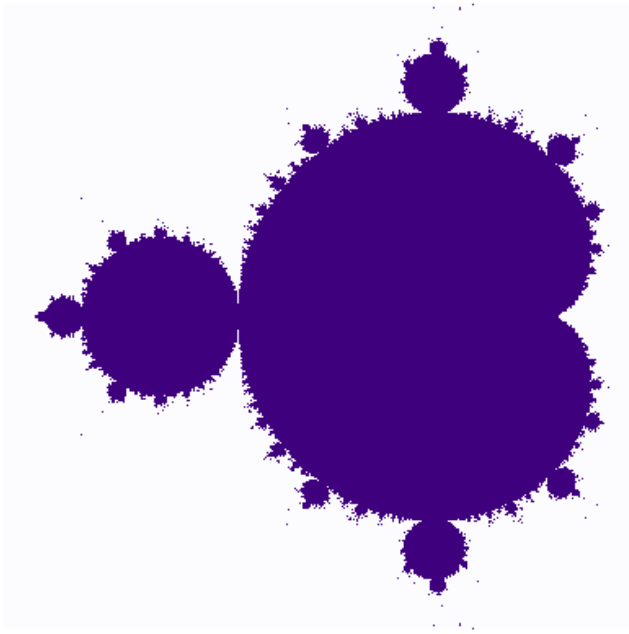
- Matrix determinant & inverse

```
>>> import numpy.linalg as la
>>> A = np.array([[1,2], [3,4]])
>>> la.det(A)
-2.0000000000000004
>>> la.inv(A)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> A @ la.inv(A)
array([[  1.00000000e+00,   0.00000000e+00],
       [  8.88178420e-16,   1.00000000e+00]])
```

Challenge: the Mandelbrot set

- Use `numpy` arrays, with universal functions/vectorization and broadcasting, to make an image of the Mandelbrot set.



- The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when repeatedly iterated from $z = 0$.