NumPy and friends

```python
import numpy as np
```

- NumPy package is *the* key building block of the Python scientific ecosystem.
- **Assume that what you want to achieve *can* be achieved in a highly optimised way within the existing framework**
- Only resort to your own solution if this is not the case
- Many resources for learning NumPy online (see links in notes)

Arrays

- Fundamental object in NumPy is *Array* (or `ndarray`), multidimensional version of a `list`
- In plain old Python a matrix would be a list of lists.

```python
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

- `data[i]` represents each row:

```python
data[1]
```

```
[4, 5, 6]
```

- To multiply every element by a number I would do something like this:

```
for row in data:
    for j, _ in enumerate(row):
        row[j] *= 2
data
```

```
[[2, 4, 6], [8, 10, 12], [14, 16, 18], [20, 22, 24]]
```

- ***Don't do this***
- NumPy is made for tasks like this with *minimum code* and *maximum efficiency*

- First create data as array
- Numerous NumPy functions produce arrays
- Simplest is numpy.array: takes data in "Pythonic" list-of-lists(-of-lists-of… etc.) form and produces `ndarray`

```
my_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
type(my_array)
```

```
numpy.ndarray
```

- Multiply array by number? Easy!

```
2 * my_array
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18],
       [20, 22, 24]])
```

- It even prints nicely

```
print(my_array)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Indexing

```python
my_array
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

- Arrays can be indexed, similar to lists

```python
print(my_array[0], my_array[1], my_array[3][1])
```

```
[1 2 3] [4 5 6] 11
```

- Better syntax for the last one

```python
my_array[3,1]
```

```
11
```

```python
my_array
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

- Also have a generalization of the _slice syntax

```python
my_array[1:, 1:]
```

```
array([[ 5,  6],
       [ 8,  9],
       [11, 12]])
```

- Slicing can be mixed with integer indexing

```
my_array[1:, 1]
```

```
array([ 5,  8, 11])
```

- NumPy has all sorts of fancy indexing options
- Indexing with integer arrays, with boolean arrays, etc.
- See the documentation

Shape

- A fundamental property of an array is `shape`:

```
# [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
my_array.shape
```

```
(4, 3)
```

- First a number of `[` corresponding to the **rank** of the array (two in the above example)
- Then number of entries giving rightmost (innermost) dimension in shape before closing `]` (3 here)
- After a number of 1D arrays `[...]` equal to the next innermost dimension (4 here), we have another closing `]`, and so on

---

- Slicing does not change the array rank

```
my_array[1:, 1:].shape
```

```
(3, 2)
```

- Integer indexing does

```
my_array[1:, 1].shape
```

```
(3,)
```

- **Note:** `(3,)` is *tuple* giving the shape while `(3)` is just the number 3 in brackets

Other ways to make arrays

- NumPy has lots of methods to create arrays

```
np.zeros((2,2))
```

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

```
np.ones((2,2))
```

```
array([[1., 1.],
       [1., 1.]])
```

```
np.full((2,2), 5)
```

```
array([[5, 5],
       [5, 5]])
```

```
np.random.random((2,2)) # random numbers uniformly in [0.0, 1.0)
```

```
array([[0.77768106, 0.8841984 ],
       [0.29779833, 0.13796426]])
```

```
np.eye(2) # Identity matrix
```

```
array([[1., 0.],
       [0., 1.]])
```

Shape shifting

- numpy.reshape to change the shape of an array
- numpy.expand_dims to insert new axes of length one.
- numpy.squeeze (the opposite) to remove new axes of length one.

- Example of `reshape`

```
my_array.reshape(2, 2, 3)
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

- Only works if the shapes are compatible. Here it's OK because the original shape was $(4, 3)$ and $4 \times 3 = 2 \times 2 \times 3$
- If shapes aren't compatible, we'll get an error

```
my_array.reshape(2, 3, 3)
```

```
ValueError: cannot reshape array of size 12 into shape (2,3,3)
```

`dtype`

- Arrays have `dtype` property that gives datatype
- If array was created from data, this will be inferred

```
my_array.dtype
```

```
dtype('int64')
```

- Functions constructing arrays have optional `dtype`

```
my_float_array = np.array([1,2,3], dtype=np.float64)
my_float_array.dtype
```

```
dtype('float64')
```

- Importantly, complex numbers are supported

```
my_float_array = np.array([1.1 + 2.3j,2.2,3.6])
my_float_array.dtype
```

```
dtype('complex128')
```

Examples of array-like data

- Position, velocity, or acceleration of particle will be three dimensional vectors, so have shape `(3,)`
- With N particles could use a $3N$ dimensional vector
- Better: an array of shape `(N,3)`. First index indexes particle number and second particle coordinate.
- $N \times M$ matrix has shape `(N,M)`
- Riemann curvature tensor in General Relativity R_{abcd} has shape `(4,4,4,4)`

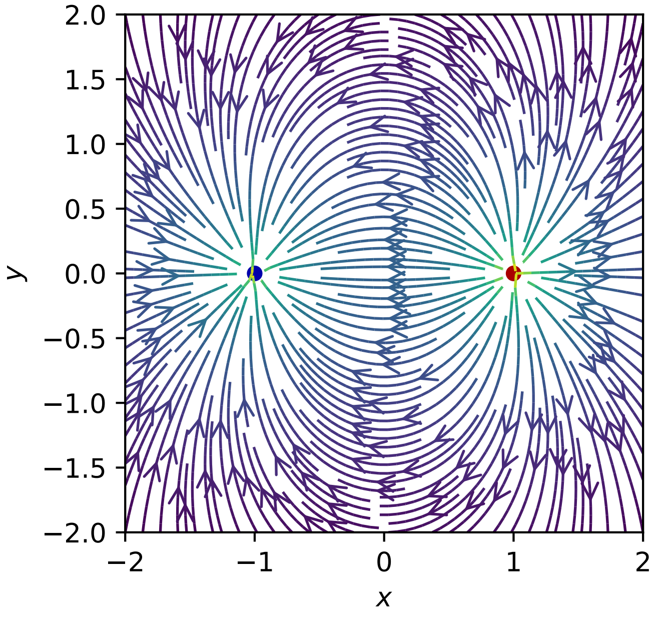
- *Fields* are functions of space and time e.g. the electric potential $\phi(\mathbf{r}, t)$
- Approximate these using a grid of space-time points $N_x \times N_y \times N_z \times N_t$
- Scalar field can be stored in an array of shape `(N_x,N_y,N_z,N_t)`
- A vector field like $\mathbf{E}(\mathbf{r}, t)$ would be `(N_x,N_y,N_z,N_t,3)`

- Very useful method to create a grid of coordinate values

```
# Grid of x, y points
nx, ny = 64, 64
x = np.linspace(-2, 2, nx)
```

```
y = np.linspace(-2, 2, ny)
X, Y = np.meshgrid(x, y)
X.shape
```

```
(64, 64)
```



Mathematical operations with arrays

- On lists

```
2 * [1, 2, 3]
```

```
[1, 2, 3, 1, 2, 3]
```

- In numerical applications what we really want is

```
2 * np.array([1, 2, 3])
```

```
array([2, 4, 6])
```

- General feature of NumPy: **all mathematical operations are performed elementwise on arrays!**

```
print(np.array([1, 2, 3]) + np.array([4, 5, 6]))
```

```
[5 7 9]
```

```
print(np.array([1, 2, 3])**2)
```

```
[1 4 9]
```

```
print(np.sqrt(np.array([1, 2, 3])))
```

```
[1.         1.41421356 1.73205081]
```

- Avoids need to write nested loops
- Loops are still there, but written in C
- This style of code is often described as *vectorized*
- In NumPy-speak vectorized functions are called *ufuncs*
- As a basic principle **never use a Python loop to access your data in NumPy code**

Broadcasting…

- …is a powerful protocol for combining arrays of different shapes, generalizing this kind of thing

```
np.array([1, 2, 3]) + 2.3
```

```
array([3.3, 4.3, 5.3])
```

- Elementwise operations performed on two arrays of same rank if, in each dimension, sizes *either match or one array has size 1*

```python
# These have shape (2, 3) and (1, 3)
np.array([[1, 2, 3], [4, 5, 6]]) + np.array([[4, 3, 2]])
```

```
array([[5, 5, 5],
       [8, 8, 8]])
```

- We can simplify this last example

```python
# These have shape (2, 3) and (3,)
np.array([[1, 2, 3], [4, 5, 6]]) + np.array([4, 3, 2])
```

```
array([[5, 5, 5],
       [8, 8, 8]])
```

- Recall example of an N-particle system described by a position array of shape `(N,3)`
- If we want to shift the entire system by a vector, just add a vector of shape `(3,)` and broadcasting will ensure that this applied correctly to each particle.

Broadcasting two arrays follows these rules:

1. If arrays do not have same rank, prepend shape of lower rank array with 1s until both shapes have same length

---

2. Two arrays are said to be *compatible* in a dimension if they have same size in that dimension, or if one of the arrays has size 1 in that dimension

---

3. Arrays can be broadcast together if they are compatible in all dimensions. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays

---

4. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

- The documentation has more detail
- Broadcasting takes some time to get used to but is immensely powerful!

Plotting with Matplotlib

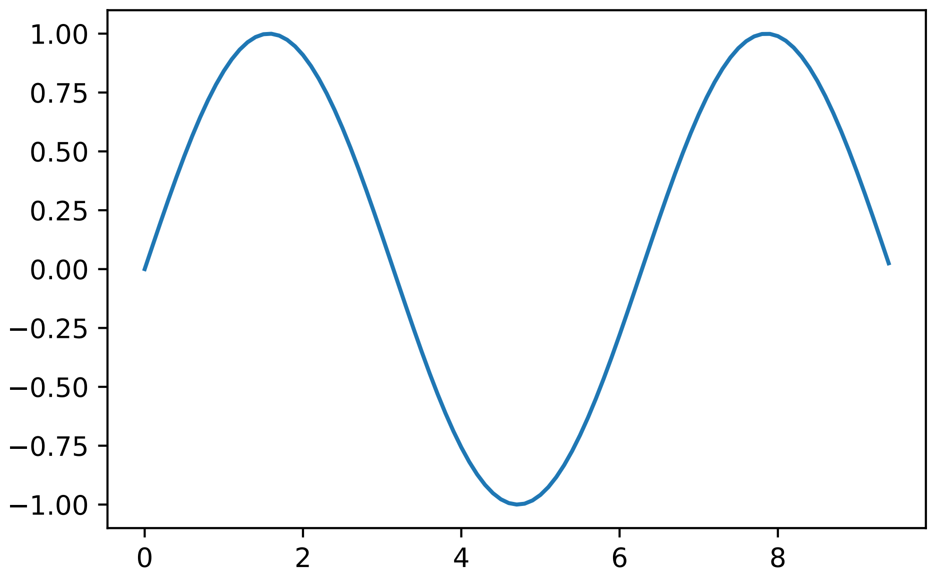
- Various specialized Python plotting libraries
- "entry-level" option is Matplotlib
- `pyplot` module provides a plotting system that is similar to MATLAB (I'm told)

```
import matplotlib.pyplot as plt
```

- Probably the second most common import you will make!

- Here's a simple example of `plot` function

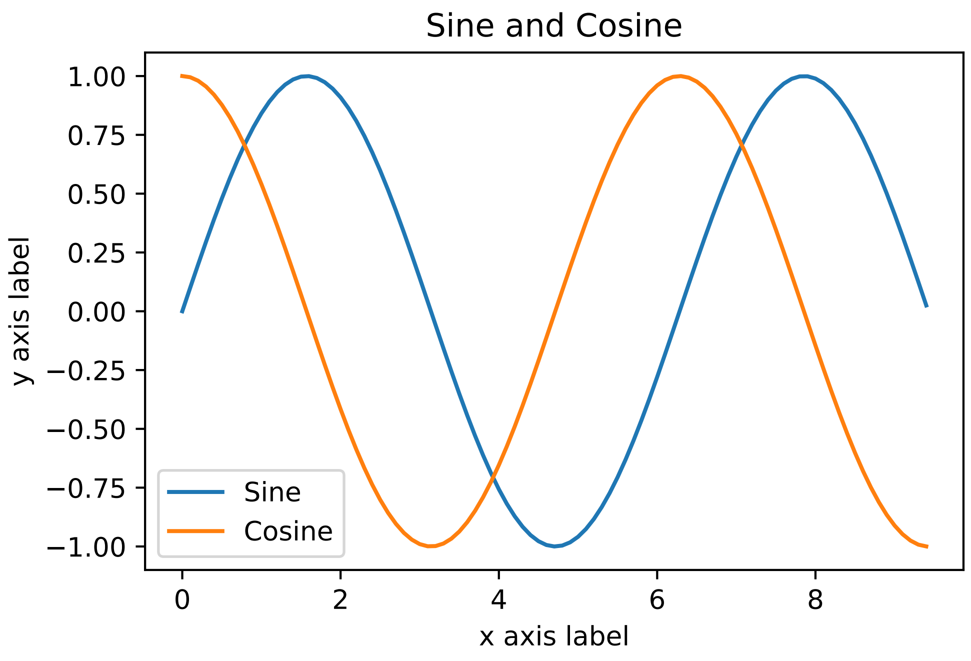
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```



- **Note**: you must call plt.show() to make graphics appear

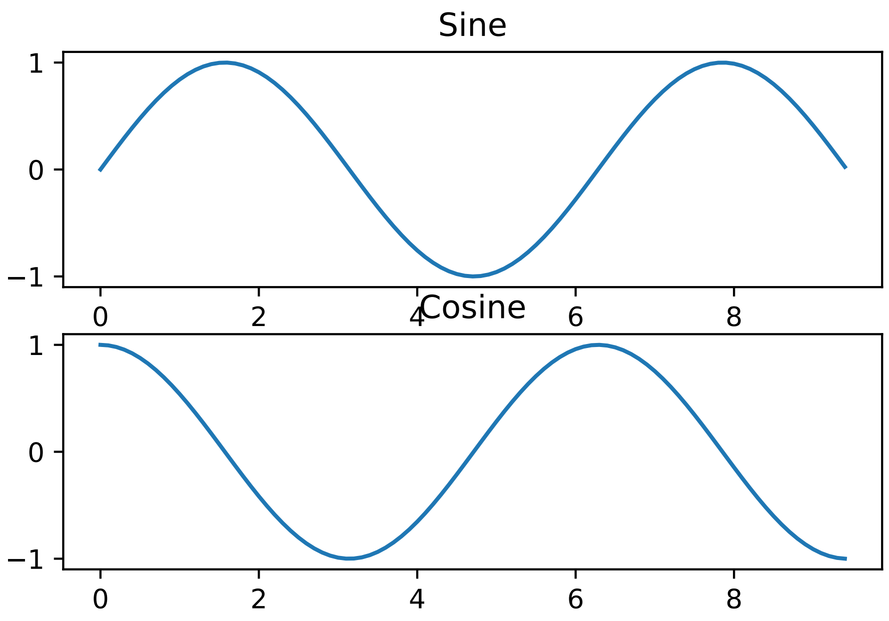
- Fancier example with some labelling

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



- Often you'll want to make several related plots and present them together

```
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

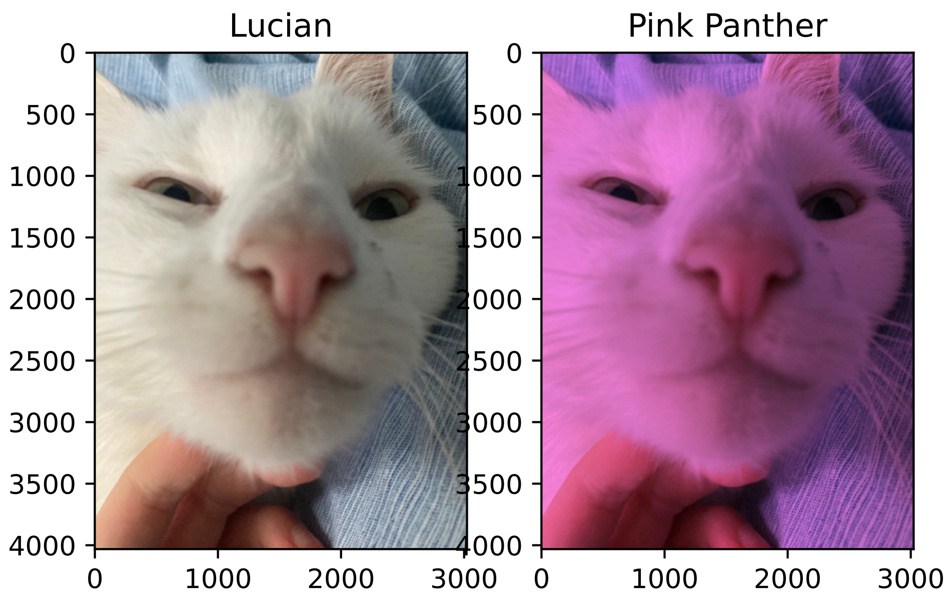


Example: playing with images

- Pixels in an image encoded as a triple of RGB values in the range [0,255] i.e. 8 bits of type `uint8` (the "u" is for "unsigned")
- Tinting an image gives a nice example of broadcasting

```
img = plt.imread('../assets/lucian.jpeg')

img_tinted = img * [1, 0.55, 1]

# Show the original image
```

```
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title("Lucian")

# Show the tinted image
plt.subplot(1, 2, 2)
plt.title("Pink Panther")
# Having multiplied by floats,
# we must cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))

plt.show()
img.shape, img.dtype
```



```
((4032, 3024, 3), dtype('uint8'))
```

This is a standard 12 megapixel image

Saving and loading data

- At some point you'll probably want to save and load data
- NumPy comes with its own save and load functions and associated binary format `.npy`
- The benefit of using these is that after loading you get back a NumPy array ready to be used

---

- A related function savez allows several arrays to be saved and then loaded as a dictionary-like object.

```
random_matrix_1 = np.random.rand(4, 4)
random_matrix_2 = np.random.rand(4, 4)
np.savez("mydata/my-matrices", first_matrix=random_matrix_1, second_matrix=random_matrix_2
%ls mydata
```

```
my-matrices.npz
```

```
my_matrix_file = np.load("mydata/my-matrices.npz")
my_matrix_file['first_matrix']
```

```
array([[0.19029225, 0.29284832, 0.65835027, 0.66609004],
       [0.38607313, 0.96309311, 0.05537036, 0.20878955],
       [0.65063472, 0.03772137, 0.0557475 , 0.6779514 ],
       [0.07950753, 0.33607414, 0.05383134, 0.16126465]])
```