INTRODUCTION TO NUMPY

WHAT IS NUMPY? (1/2)

- A library to manipulate arrays
- An array is an indexable collection of items of the same type
    - Indexable means we can fetch specific parts of the array by their location
- Arrays are mutable by default -- we can modify values
    - New values need to be compatible with the type
    - Can make them immutable (use `.setflags(write=False)` on an array)

WHAT IS NUMPY? (2/2)

- NumPy makes manipulating arrays fast and easy
- Apply mathematical operations over specific dimensions of multi-dimensional arrays
- Support for many data types
- Many scientific packages are built on top of it (e.g. Pandas)

NUMPY ARRAYS

- NumPy arrays can have as many dimensions as you like, for example, an array could be:
    - a vector with n elements
    - a matrix with m rows and n columns (m, n)
    - a tensor of size (m, n, p)
- The most common way to create a NumPy array is from a standard Python list

PRACTICAL INTRODUCTION

```
import numpy as np

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

```
array([1, 2, 3])
```

```
a.shape
```

```
(3,)
```

SIZES, SHAPES, AND DIMENSIONS

```
a.size
```

```
3
```

```
a.ndim # number of dimensions
```

```
1
```

2-DIMENSIONAL ARRAYS

We can make a two-dimensional array (a matrix) by inputting a list of lists:

```
b = np.array([[0,1,2],[3,4,5]])
b
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
b.shape
```

```
(2, 3)
```

```
b.size
```

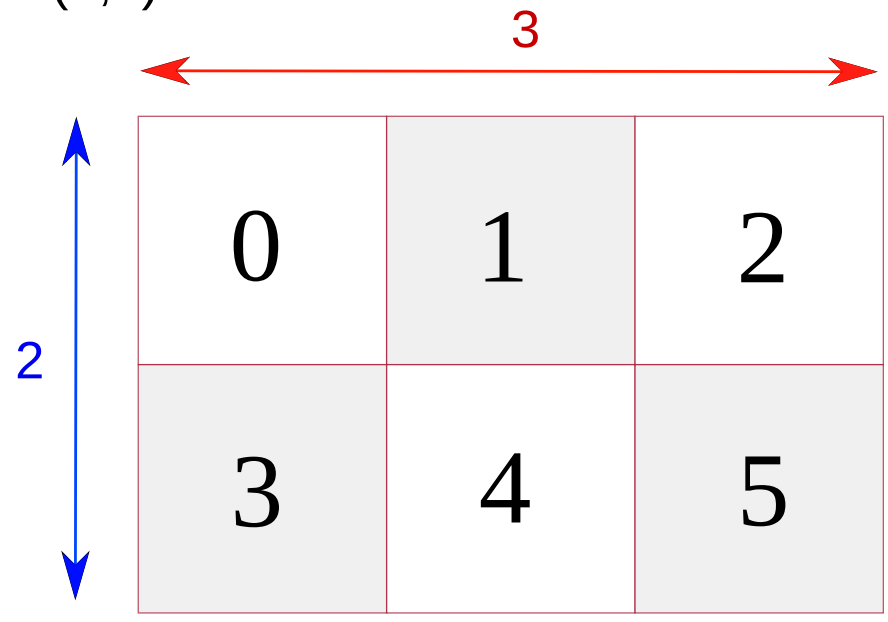
```
6
```

```
b.ndim
```

```
2
```

2-DIMENSIONAL ARRAYS

b.shape = (2,3)



N-DIMENSIONAL ARRAYS

We can continue scaling this up to as many dimensions as we would like:

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

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

       [[1, 2, 3],
        [4, 5, 6]]])
```

```
c.shape
```

```
(2, 2, 3)
```

```
c.size
```

```
12
```

```
c.ndim
```

```
3
```

RESHAPING

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

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

```
b.shape
```

```
(2, 3)
```

```
b.reshape(6)
```

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

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

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

INDEXING ARRAYS

To get the nth element of an array, we can index with `a[n]`.

- Python/NumPy index from 0
- We can index over multiple dimensions

```
a = np.array([1,2,3])
a[0]
```

```
1
```

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

```
1
```

```
b[0,2]
```

```
3
```

INDEXING ARRAYS

To index a whole dimension of the array, use `:`. For instance, array `b` has two dimensions, rows and columns.
To get the whole of row 0, we can either index as `b[0,:]` or `b[0]`

```
b[0,:]
```

```
array([1, 2, 3])
```

```
b[0]
```

```
array([1, 2, 3])
```

INDEXING ARRAYS

The columns are the second dimension, so to get the whole of column 0 we do:

```
b[:,0]
```

```
array([1, 4])
```

So the syntax `b[M,N]` indexes `b` by row `M` and column `N`.

Practical 1 - Arrays, Reshaping, Indexing

practical1.ipynb

INDEXING CONTINUED

We can index with a list too. Say we want the first, second, and fourth elements of a:

```
a = np.array([0,1,2,3,4,5])
indices = [0,1,3]
a[indices]
```

```
array([0, 1, 3])
```

BOOLEAN INDEXING

We can also do indexing by True/False statements:

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

```
array([False, False, False, False,  True,  True])
```

We can use this mask to index `a` itself:

```
a[a > 3]
```

```
array([4, 5])
```

SLICING

We can slice up arrays into sub-arrays by providing a lower and upper (exclusive) index:

```
a = np.array([0,1,2,3,4,5])

a[1:4]
```

```
array([1, 2, 3])
```

Get the first and last 3 elements of the array:

```
a[:3]
```

```
array([0, 1, 2])
```

```
a[3:]
```

```
array([3, 4, 5])
```

SLICING

You can also count back from the end of the array using a minus sign:

```
a[-3:]
```

```
array([3, 4, 5])
```

So `-3` means count back 3 indices from the end of `a`, and `:` returns all elements after this position.
How would you get all the elements before the item in position `-3`?
How about just the last element?

SLICING IN MULTIPLE DIMENSIONS

Just as with regular indexing, we can slice index over multiple dimensions:

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

# columns 0 and 1
b[:, 0:2]
```

```
array([[1, 2],
       [4, 5]])
```

```
# row 0, columns 0 and 1
b[0,0:2]
```

```
array([1, 2])
```

NUMPY FUNCTIONS

NumPy makes it easy to apply functions to arrays:

- Maths functions (`np.sin`, `np.sum`, `np.round`, `np.exp`)
- Logic (`np.equal`, `np.allclose`, `np.any`)
- Sorting, searching, counting (`np.argwhere`, `np.sort`)
- and many more!

For example:

```
a = np.array([1,2,3])
np.sum(a)
```

NUMPY FUNCTIONS

We can apply functions over specific dimensions/axes:

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

there are two dimensions/axes (rows = axis 0, columns = axis 1).
So to sum up each of the columns:

```
np.sum(b, axis=0)
```

```
array([5, 7, 9])
```

And to sum the rows:

```
np.sum(b, axis=1)
```

```
array([ 6, 15])
```

BROADCASTING

NumPy also supports broadcasting when the dimensions don't match up. For instance, to double every element in an array we could either do:

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

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

Or we could do:

```
a*2
```

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

NumPy automatically stretches out the scalar 2 to be the vector of 2s in the first example so that they can be multiplied together.

BROADCASTING

- As with all things NumPy, this scales up to multiple dimensions very easily.
- NumPy checks each aligned pair of dimensions to see if they are:
    - equal in size
    - one of them is of size one
- Which of these pairs can be broadcast?
    - 5,1?
    - 2,4?
    - 3,3?

BROADCASTING OVER MULTIPLE DIMENSIONS

For `c = a * b`, we can infer the shape of `c` from the shapes of `a` and `b`:

```
a.shape = (1 x 2)
b.shape = (5 x 2)
c.shape = (5 x 2)
```

When they have different dimensions, align the trailing dimension:

```
a.shape = (5 x 6)
b.shape =      (1)
c.shape = (5 x 6)
```

BROADCASTING OVER MULTIPLE DIMENSIONS

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

# multiply col 1 by 2, col 2 by 5, col 3 by 10
b = np.array([2,5,10])
a*b
```

```
array([[ 2, 10, 30],
       [ 8, 25, 60]])
```

MODIFYING ARRAYS

We can use indexing, slicing, and broadcasting to modify the values in arrays:

```
# indexing
a = np.array([1,2,3])
a[0] = 5
```

```
# slicing
a[0:3] = [5, 5, 5]
a
```

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

```
# broadcasting
a[0:3] = 2
a
```

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

Practical 2 - Slicing, Broadcasting, Modifying

practical2.ipynb

GENERATING NEW ARRAYS DETERMINISTICALLY

- `np.linspace(lower, upper, N)`: N linearly distributed numbers between lower and upper
- `np.ones((shape))`: an array of ones of shape `shape`
- `np.zeros((shape))`: an array of zeros of shape `shape`
- `np.arange(upper)`: integers from 0 to upper
- `np.eye(N)`: identity matrix of size $N \times N$

GENERATING NEW ARRAYS RANDOMLY

- `np.random.randn()`:
- `np.random.random()`:
- `np.random.choice(list)`: choose a number from the list

STACKING AND RESHAPING

- Stack two arrays vertically `np.vstack([a,b])` or horizontally `np.hstack([a,b])`

```
a = np.ones((4))
b = np.ones((4))

np.vstack([a,b])
```

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

```
np.hstack([a,b])
```

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

NUMPY ARRAY FUNCTIONS

There are certain functions which can be applied directly to the array (i.e. you can do `a.function()` instead of `np.function(a)`.

- `a.mean()` mean
- `a.T` : transpose
- `a.argmax()`: index of the max element of `a`

See documentation for more.

Practical 3 - Generating, Stacking, Methods

practical3.ipynb