Introducing NumPy

Python is convenient, but it can also be slow. However, it does allow you to access libraries that execute faster code written in languages like C. NumPy is one such library: it provides fast alternatives to math operations in Python and is designed to work efficiently with groups of numbers - like matrices.

NumPy is a large library and we are only going to scratch the surface of it here. If you plan on doing much math with Python, you should definitely spend some time exploring its documentation to learn more.

Importing NumPy

When importing the NumPy library, the convention you'll see used most often – including here – is to name it `np`, like so:

```
import numpy as np
```

Now you can use the library by prefixing the names of functions and types with `np.`, which you'll see in the following examples.

Data Types and Shapes

The most common way to work with numbers in NumPy is through `ndarray` objects. They are similar to Python lists, but can have any number of dimensions. Also, `ndarray` supports fast math operations,

which is just what we want.

Since it can store any number of dimensions, you can use `ndarray`s to represent any of the data types we covered before: scalars, vectors, matrices, or tensors.

Scalars

Scalars in NumPy are a bit more involved than in Python. Instead of Python's basic types like `int`, `float`, etc., NumPy lets you specify signed and unsigned types, as well as different sizes. So instead of Python's `int`, you have access to types like `uint8`, `int8`, `uint16`, `int16`, and so on.

These types are important because every object you make (vectors, matrices, tensors) eventually stores scalars. And when you create a NumPy array, you can specify the type - but **every item in the array must have the same type**. In this regard, NumPy arrays are more like C arrays than Python lists.

If you want to create a NumPy array that holds a scalar, you do so by passing the value to NumPy's `array` function, like so:

```
s = np.array(5)
```

You can still perform math between `ndarray`s, NumPy scalars, and normal Python scalars, though, as you'll see in the element-wise math

lesson.

You can see the shape of your arrays by checking their `shape` attribute. So if you executed this code:

```
s.shape
```

it would print out the result, an empty pair of parenthesis, `()`. This indicates that it has zero dimensions.

Even though scalars are inside arrays, you still use them like a normal scalar. So you could type:

```
x = s + 3
```

and `x` would now equal `8`. If you were to check the type of `x`, you'd find it is probably `numpy.int64`, because its working with NumPy types, not Python types.

By the way, even scalar types support most of the array functions. so you can call `x.shape` and it would return `()` because it has zero dimensions, even though it is not an array. If you tried that with a normal Python scalar, you'd get an error.

Vectors

To create a vector, you'd pass a Python list to the `array` function, like

this:

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

If you check a vector's `shape` attribute, it will return a single number representing the vector's one-dimensional length. In the above example, `v.shape` would return `(3,)`

Now that there is a number, you can see that the shape is a tuple with the sizes of each of the `ndarray`'s dimensions. For scalars it was just an empty tuple, but vectors have one dimension, so the tuple includes a number and a comma. (Python doesn't understand `(3)` as a tuple with one item, so it requires the comma. You can read more about tuples here)

You can access an element within the vector using indices, like this:

```
x = v[1]
```

Now `x` equals `2`.

NumPy also supports advanced indexing techniques. For example, to access the items from the second element onward, you would say:

```
v[1:]
```

and it would return an array of `[2, 3]`. NumPy slicing is quite powerful,

allowing you to access any combination of items in an `ndarray`. But it can also be a bit complicated, so you should read up on it in the documentation.

Matrices

You create matrices using NumPy's `array` function, just you did for vectors. However, instead of just passing in a list, you need to supply a list of lists, where each list represents a row. So to create a 3x3 matrix containing the numbers one through nine, you could do this:

```
m = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

Checking its `shape` attribute would return the tuple `(3, 3)` to indicate it has two dimensions, each length 3.

You can access elements of matrices just like vectors, but using additional index values. So to find the number `6` in the above matrix, you'd access `m[1][2]`.

Tensors

Tensors are just like vectors and matrices, but they can have more dimensions. For example, to create a 3x3x2x1 tensor, you could do the following:

```
t = np.array([[[[1],[2]],[[3],[4]],[[5],[6]]],[[[7],[8]],\
    [[9],[10]],[[11],[12]]],[[[13],[14]],[[15],[16]],[[17],
[17]]]])
```

And `t.shape` would return `(3, 3, 2, 1)`.

You can access items just like with matrices, but with more indices. So `t[2][1][1][0]` will return `16`.

Changing Shapes

Sometimes you'll need to change the shape of your data without actually changing its contents. For example, you may have a vector, which is one-dimensional, but need a matrix, which is two-dimensional. There are two ways you can do that.

Let's say you have the following vector:

```
v = np.array([1,2,3,4])
```

Calling `v.shape` would return `(4,)`. But what if you want a 1x4 matrix? You can accomplish that with the `reshape` function, like so:

```
x = v.reshape(1,4)
```

Calling `x.shape` would return `(1,4)`. If you wanted a 4x1 matrix, you could do this:

```
x = v.reshape(4,1)
```

The `reshape` function works for more than just adding a dimension of size 1. Check out its documentation for more examples.

One more thing about reshaping NumPy arrays: if you see code from experienced NumPy users, you will often see them use a special slicing syntax instead of calling `reshape`. Using this syntax, the previous two examples would look like this:

```
x = v[None, :]
```

or

```
x = v[:, None]
```

Those lines create a slice that looks at all of the items of `v` but asks NumPy to add a new dimension of size 1 for the associated axis. It may look strange to you now, but it's a common technique so it's good to be aware of it.