Transpose

Getting the transpose of a matrix is really easy in NumPy. Simply access its `T` attribute. There is also a `transpose()` function which returns the same thing, but you'll rarely see that used anywhere because typing `T` is so much easier. :)

For example:

```
m = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
m
# displays the following result:
# array([[ 1,  2,  3,  4],
#        [ 5,  6,  7,  8],
#        [ 9, 10, 11, 12]])

m.T
# displays the following result:
# array([[ 1,  5,  9],
#        [ 2,  6, 10],
#        [ 3,  7, 11],
#        [ 4,  8, 12]])
```

NumPy does this without actually moving any data in memory - it simply changes the way it indexes the original matrix - so it's quite efficient.

However, that also means you need to be careful with how you modify objects, because **they are sharing the same data**. For example, with the same matrix `m` from above, let's make a new variable `m_t` that stores `m`'s transpose. Then look what happens if we modify a value in `m_t`:

```
m_t = m.T
m_t[3][1] = 200
m_t
# displays the following result:
# array([[ 1,   5, 9],
#        [ 2,   6, 10],
#        [ 3,   7, 11],
#        [ 4, 200, 12]])

m
# displays the following result:
# array([[ 1,  2,  3,   4],
#        [ 5,  6,  7, 200],
#        [ 9, 10, 11,  12]])
```

Notice how it modified both the transpose and the original matrix, too! That's because they are sharing the same copy of data. So remember to consider the transpose just as a different view of your matrix, rather than a different matrix entirely.

A real use case

I don't want to get into too many details about neural networks because you haven't covered them yet, but there is one place you will almost certainly end up using a transpose, or at least thinking about it.

Let's say you have the following two matrices, called `inputs` and `weights`,

```
inputs = np.array([[-0.27,  0.45,  0.64, 0.31]])
inputs
# displays the following result:
# array([[-0.27,  0.45,  0.64,  0.31]])

inputs.shape
# displays the following result:
# (1, 4)

weights = np.array([[0.02, 0.001, -0.03, 0.036], \
    [0.04, -0.003, 0.025, 0.009], [0.012, -0.045, 0.28, -0.067]])

weights
# displays the following result:
# array([[ 0.02 ,  0.001, -0.03 ,  0.036],
#        [ 0.04 , -0.003,  0.025,  0.009],
#        [ 0.012, -0.045,  0.28 , -0.067]])

weights.shape
# displays the following result:
# (3, 4)
```

I won't go into what they're for because you'll learn about them later, but you’re going to end up wanting to find the **matrix product** of these two matrices.

If you try it like they are now, you get an error:

```
np.matmul(inputs, weights)
# displays the following error:
# ValueError: shapes (1,4) and (3,4) not aligned: 4 (dim 1) != 3 (dim 0)
```

If you did the matrix multiplication lesson, then you’ve seen this error before. It’s complaining of incompatible shapes because the number of columns in the left matrix, `4`, does not equal the number of rows in the right matrix, `3`.

So that doesn’t work, but notice if you take the transpose of the `weights` matrix, it will:

```
np.matmul(inputs, weights.T)
# displays the following result:
# array([[-0.01299,  0.00664,  0.13494]])
```

It also works if you take the transpose of `inputs` instead and swap their order, like we showed in the video:

```
np.matmul(weights, inputs.T)
# displays the following result:
# array([[-0.01299],#
#        [ 0.00664],
#        [ 0.13494]])
```

The two answers are transposes of each other, so which multiplication you use really just depends on the shape you want for the output.