Element-wise operations

The Python way

Suppose you had a list of numbers, and you wanted to add `5` to every item in the list. Without NumPy, you might do something like this:

```
values = [1,2,3,4,5]
for i in range(len(values)):
    values[i] += 5

# now values holds [6,7,8,9,10]
```

That makes sense, but it's a lot of code to write and it runs slowly because it's pure Python.

Note: Just in case you aren't used to using operators like `+=`, that just means "add these two items and then store the result in the left item." It is a more succinct way of writing `values[i] = values[i] + 5`. The code you see in these examples makes use of such operators whenever possible.

The NumPy way

In NumPy, we could do the following:

```
values = [1,2,3,4,5]
values = np.array(values) + 5

# now values is an ndarray that holds [6,7,8,9,10]
```

Creating that array may seem odd, but normally you'll be storing your data in `ndarray`s anyway. So if you already had an `ndarray` named `values`, you could have just done:

```
values += 5
```

We should point out, NumPy actually has functions for things like adding, multiplying, etc. But it also supports using the standard math operators. So the following two lines are equivalent:

```
x = np.multiply(some_array, 5)
x = some_array * 5
```

We will usually use the operators instead of the functions because they are more convenient to type and easier to read, but it's really just personal preference.

One more example of operating with scalars and `ndarrays`. Let's say you have a matrix `m` and you want to reuse it, but first you need to set all its values to zero. Easy, just multiply by zero and assign the result back to the matrix, like this:

```
m *= 0

# now every element in m is zero, no matter how many dimensions it has
```

Element-wise Matrix Operations

The same functions and operators that work with scalars and matrices also work with other dimensions. You just need to make sure that the items you perform the operation on have compatible shapes.

Let's say you want to get the squared values of a matrix. That's simply `x = m * m` (or if you want to assign the value back to `m`, it's just `m *= m`

This works because it's an element-wise multiplication between two identically-shaped matrices. (In this case, they are shaped the same because they are actually the same object.)

Here's the example from the video:

```
a = np.array([[1,3],[5,7]])
a
# displays the following result:
# array([[1, 3],
#        [5, 7]])

b = np.array([[2,4],[6,8]])
b
# displays the following result:
# array([[2, 4],
#        [6, 8]])

a + b
# displays the following result
#      array([[ 3,  7],
#             [11, 15]])
```

And if you try working with incompatible shapes, like the other example from the video, you'd get an error:

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

a.shape
# displays the following result:
#  (2, 2)

c.shape
# displays the following result:
#  (3, 3)

a + c
# displays the following error:
# ValueError: operands could not be broadcast together with shapes (2,2) (3,3)
```

You'll learn more about what that "could not be broadcast together" means in a later lesson, but for now, just notice that the two shapes are different so we can't perform the element-wise operation.

NEXT