Numpy

Numpy is a popular library in Python for performing lots of data analysis tasks, because it provides data structures for n-dimensional arrays and matrices. These structures support many of the common operations you might want to do on a matrix.

Let's start by creating an array of 9 random integers in the range [-10, 10)

In [2]:
```
import numpy as np

data = np.random.randint(-10, 10, size = 9)
print(data)
```

```
[-9 -3 -8 -7  4 -4  2  2 -2]
```

Note two things here. First, while this `data` looks like a list, it isn't. It is an `ndarray`, which stands for *n-dimensional array*. This is the main datatype that numpy provides.

For any ndarray, we can ask for its *shape*: this tells us how many dimensions it has, and how big each dimension is. In this case, we have 1 dimension, and it has 9 elements.

In [3]:
```
print(type(data))
print(data.shape)
```

```
<class 'numpy.ndarray'>
(9,)
```

Reshaping

One of the most useful abilities in Numpy is the ability to *reshape* one ndarray into another. Think of this as "pouring" the data from one array, row by row, into the next array, row by row. So, for example, we can reshape the 9-element, 1-dimensional array into a 9x1 2-dimensional array:

In [4]:
```
data_arr = np.reshape(data, (9, 1))
print(data_arr)
print(data_arr.shape)
print(type(data_arr))
```

```
[[-9]
 [-3]
 [-8]
 [-7]
 [ 4]
 [-4]
 [ 2]
 [ 2]
 [-2]]
(9, 1)
<class 'numpy.ndarray'>
```

Or into a 1x9 2-dimsional array:

In [5]:
```
data_arr = np.reshape(data_arr, (1, 9))
print(data_arr)
print(data_arr.shape)
```

```
[[-9 -3 -8 -7  4 -4  2  2 -2]]
(1, 9)
```

Or into a 3x3. Note that in this case, the data is filled in row-by-row:

In [6]:
```
data_arr = np.reshape(data_arr, (3, 3))
print(data_arr)
print(data_arr.shape)
```

```
[[-9 -3 -8]
 [-7  4 -4]
 [ 2  2 -2]]
(3, 3)
```

Note that you can only reshape ndarrays into "compatible" ones: they must be able to hold exactly the same amount of data. Reshaping the array into one that is too small or too large won't work:

In [7]:
```
too_large = np.reshape(data_arr, (4, 4))
too_small = np.reshape(data_arr, (3, 2))
```

```
---------------------------------------------------------------
-----------------
ValueError                                Traceback (most
recent call last)
<ipython-input-7-a7be57914bd9> in <module>
----> 1 too_large = np.reshape(data_arr, (4, 4))
      2 too_small = np.reshape(data_arr, (3, 2))

/usr/local/lib/python3.7/site-packages/numpy/core/fromnume
ric.py in reshape(a, newshape, order)
    290            [5, 6]])
    291     """
--> 292     return _wrapfunc(a, 'reshape', newshape, order
=order)
    293
    294

/usr/local/lib/python3.7/site-packages/numpy/core/fromnume
ric.py in _wrapfunc(obj, method, *args, **kwds)
     54 def _wrapfunc(obj, method, *args, **kwds):
     55     try:
---> 56         return getattr(obj, method)(*args, **kwds)
     57
     58     # An AttributeError occurs if the object does
not have

ValueError: cannot reshape array of size 9 into shape (4,4
)
```

Collective operations:

One useful thing that Numpy supports is the ability to do "collective" operations on the rows/columns/etc of an n-dimensional array. For example, you can compute the mean of every column in `data_arr` by asking for the mean along axis 0:

In [8]:
```
column_mean = np.mean(data_arr, axis=0)
print(column_mean)
```

```
[-4.66666667  1.         -4.66666667]
```

Note that this creates a 1x3 ndarray: each column has its own mean, so there are as many entries in the result as there are columns in the original.

You can also compute means along other dimensions, such as along the row:

In [9]:
```
row_mean = np.mean(data_arr, axis=1)
print(row_mean)
```

```
[-6.66666667 -2.33333333  0.66666667]
```

You can also compute things like standard deviation and variance:

In [10]:
```
np.std(data_arr, axis=0)
```

Out[10]:
```
array([4.78423336, 2.94392029, 2.49443826])
```

In [11]:
```
np.var(data_arr, axis = 0)
```

Out[11]:
```
array([22.88888889,  8.66666667,  6.22222222])
```

In [12]:
```
np.sum(data_arr, axis = 0)
```

Out[12]:

```
array([-14,   3, -14])
```

nd-array math

One of the trickier things to get used to in numpy is how math on ndarrays is performed.

The first thing to keep in mind is that Numpy *always* tries to do element-by-element operations if it can. If you add two arrays together, you will add together the individual elements if the shapes are compatible. But, surprisingly, the same thing will happen if you *multiply* two arrays together:

In [15]:
```
data_tmp = np.random.randint(-5, 5, (3, 3))
print(data_tmp)
```

```
[[-5  0  0]
 [ 2 -1 -4]
 [-5 -5  1]]
```

In [16]:
```
print(data_arr)
print(data_arr + data_tmp)
print(data_arr * data_tmp)
```

```
[[-9 -3 -8]
 [-7  4 -4]
 [ 2  2 -2]]
[[-14  -3  -8]
 [ -5   3  -8]
 [ -3  -3  -1]]
[[ 45   0   0]
 [-14  -4  16]
 [-10 -10  -2]]
```

Broadcasting

So what if you try to do operations on ndarrays that aren't the same size? Some times, the operation will just fail:

In [19]:
```
data_bad = np.array([[2, 3], [5, 6]])
print(data_bad)
```

```
[[2 3]
 [5 6]]
```

In [20]:
```
data_arr + data_bad
```

```
---------------------------------------------------------------
-----------------
ValueError                                Traceback (most
recent call last)
<ipython-input-20-572e82ec1a4a> in <module>
----> 1 data_arr + data_bad

ValueError: operands could not be broadcast together with
shapes (3,3) (2,2)
```

But other times, numpy will try to "broadcast" the operands so that the dimensions line up. The way it does this is by copying the data along missing dimensions (or along dimensions of size 1) to create compatible ndarrays. The rules of broadcasting (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html) are complicated, so here we'll just talk about a few of them.

If you try to perform math with a scalar value, numpy will copy that value out into an array of matching dimension before performing the operation:

In [21]:
```
print(data_arr)
print(data_arr - 1)
```

```
[[-9 -3 -8]
 [-7  4 -4]
 [ 2  2 -2]]
[[-10  -4  -9]
 [ -8   3  -5]
 [  1   1  -3]]
```

If instead you have a column vector of the same size as the other operand, numpy will copy that column enough times to match the total number of columns:

In [22]:
```
col_data = np.random.randint(-5, 5, size=(3, 1))
print(col_data)
```

```
[[ 4]
 [-4]
 [ 2]]
```

In [23]:
```
print(data_arr)
print(data_arr - col_data)
```

```
[[-9 -3 -8]
 [-7  4 -4]
 [ 2  2 -2]]
[[-13  -7 -12]
 [ -3   8   0]
 [  0   0  -4]]
```

And the same for row vectors:

In [24]:
```
row_data = np.random.randint(-5, 5, size=(1, 3))
print(row_data)
```

```
[[3 3 4]]
```

In [25]:
```
print(data_arr)
print(data_arr - row_data)
```

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

Remember that numpy does element-by-element math, so multiplication does not do matrix multiplication:

In [26]:
```
print(data_arr * data_tmp)
```

```
[[ 45   0   0]
 [-14  -4  16]
 [-10 -10  -2]]
```

Instead, you need to use `numpy.dot`:

In [28]:
```
print(np.dot(data_arr, data_tmp))
```

```
[[ 79  43   4]
 [ 63  16 -20]
 [  4   8 -10]]
```

Numpy Matrices

While you can use ndarrays to do a lot of matrix math (e.g., `numpy.dot` for matrix multiplication, `numpy.linalg.inv` for matrix inversion, etc. -- see Numpy Linear Algebra (https://docs.scipy.org/doc/numpy/reference/routines.linalg.html)) numpy also provides a special class for doing matrix math called, unsurprisingly, `Matrix`.

You can create a matrix by passing a two-dimensional ndarray to `numpy.matrix`:

In [30]:
```
data_mtx = np.matrix(data_arr)
print(data_arr)
print(data_mtx)
```

```
[[-9 -3 -8]
 [-7  4 -4]
 [ 2  2 -2]]
[[-9 -3 -8]
 [-7  4 -4]
 [ 2  2 -2]]
```

In [31]:
```
new_mtx = np.matrix(data_tmp)
```

And now matrix multiplication works exactly like you expect it to:

In [33]:
```
print(data_mtx * new_mtx)
```

```
[[ 79  43   4]
 [ 63  16 -20]
 [  4   8 -10]]
```

Matrices also give easy access to Transpose operations and Inverse operations:

In [34]:
```
print(data_mtx.T) #Transpose
print(data_mtx.I) #Inverse
```

```
[[-9 -7  2]
 [-3  4  2]
 [-8 -4 -2]]
[[-0.         -0.09090909  0.18181818]
 [-0.09090909  0.14049587  0.08264463]
 [-0.09090909  0.04958678 -0.23553719]]
```