*3 - Introduction to NumPy

June 19, 2016

As a reminder, one of the prerequisites for this course if programming experience, especially in Python. If you do not have experience in Python specifically, we strongly recommend you go through the Codecademy Python course as soon as possible to brush up on the basics of Python.

Before going through this notebook, you may want to take a quick look at 7 - Debugging.ipynb if you haven't already for some tips on debugging your code when you get stuck.

We will be making heavy use of the Python library called NumPy. It is not included by default, so we first need to import it. Go ahead and run the following cell:

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

Now, we have access to all NumPy functions via the variable `np` (this is the convention in the Scientific Python community for referring to NumPy). We can take a look at what this variable actually is, and see that it is in fact the `numpy` module (remember that you will need to have run the cell above before `np` will be defined!):

```
In [ ]: np
```

NumPy is incredibly powerful and has many features, but this can be a bit intimidating when you're first starting to use it. If you are familiar with other scientific computing languages, the following guides may be of use: * NumPy for Matlab Users: http://mathesaurus.sourceforge.net/matlab-numpy.html * NumPy for R (and S-Plus) Users: http://mathesaurus.sourceforge.net/r-numpy.html

If not, don't worry! Here we'll go over the most common NumPy features.

0.1 Arrays and lists

The core component of NumPy is the `ndarray`, which is pronounced like "N-D array" (i.e., 1-D, 2-D, . . . , N-D). We'll use both the terms `ndarray` and "array" interchangeably. For now, we're going to stick to just 1-D arrays – we'll get to multidimensional arrays later.

Arrays are very similar to `lists`. Let's first review how lists work. Remember that we can create them using square brackets:

```
In [ ]: mylist = [3, 6, 1, 0, 10, 3]
        mylist
```

And we can access an element via its index. To get the first element, we use an index of 0:

```
In [ ]: print("The first element of 'mylist' is: " + str(mylist[0]))
```

To get the second element, we use an index of 1:

```
In [ ]: print("The second element of 'mylist' is: " + str(mylist[1]))
```

And so on.

Arrays work very similarly. The first way to create an array is from an already existing list:

```
In [ ]: myarray = np.array(mylist) # equivalent to np.array([3, 6, 1, 0, 10, 3])
        myarray
```

Notice that myarray looks different than mylist – it actually tells you that it's an array. If we take a look at the types of mylist and myarray, we will also see that one is a list and one is an array. Using type can be a very useful way to verify that your variables contain what you want them to contain:

```
In [ ]: # look at what type mylist is
        type(mylist)
```

```
In [ ]: # look at what type myarray is
        type(myarray)
```

We can get elements from a NumPy array in exactly the same way as we get elements from a list:

```
In [ ]: print("The first element of 'myarray' is: " + str(myarray[0]))
        print("The second element of 'myarray' is: " + str(myarray[1]))
```

0.2 Array slicing

Also like lists, we can use "slicing" to get different parts of the array. Slices look like myarray[a:b:c], where a, b, and c are all optional (though you have to specify at least one). a is the index of the beginning of the slice, b is the index of the end of the slice (exclusive), and c is the step size.

Note that the exclusive slice indexing described above is different than some other languages you may be familiar with, like Matlab and R. `myarray[1:2]` returns only the second elment in myarray in Python, instead of the first and second element.

First, let's quickly look at what is in our array and list (defined above), for reference:

```
In [ ]: print("mylist:", mylist)
        print("myarray:", myarray)
```

Now, to get all elements except the first:

```
In [ ]: myarray[1:]
```

To get all elements except the last:

```
In [ ]: myarray[:-1]
```

To get all elements except the first and the last:

```
In [ ]: myarray[1:-1]
```

To get every other element of the array (beginning from the first element):

```
In [ ]: myarray[::2]
```

To get every element of the array (beginning from the second element):

```
In [ ]: myarray[1::2]
```

And to reverse the array:

```
In [ ]: myarray[::-1]
```

0.3 Array computations

So far, NumPy arrays seem basically the same as regular lists. What's the big deal about them?

0.3.1 Working with single arrays

One advantage of using NumPy arrays over lists is the ability to do a computation over the entire array. For example, if you were using lists and wanted to add one to every element of the list, here's how you would do it:

```
In [ ]: mylist = [3, 6, 1, 0, 10, 22]
        mylist_plus1 = []
        for x in mylist:
            mylist_plus1.append(x + 1)
        mylist_plus1
```

Or, you could use a list comprehension:

```
In [ ]: mylist = [3, 6, 1, 0, 10, 22]
        mylist_plus1 = [x + 1 for x in mylist]
        mylist_plus1
```

If you haven't seen list comprehensions before, we strongly recommend that you go through the "Advanced Topics" section on Codecademy before proceeding!:

In contrast, adding one to every element of a NumPy array is far simpler:

```
In [ ]: myarray = np.array([3, 6, 1, 0, 10, 22])
        myarray_plus1 = myarray + 1
        myarray_plus1
```

This won't work with normal lists. For example, if you ran `mylist + 1`, you'd get an error like this:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-5b3951a16990> in <module>()
----> 1 mylist + 1

TypeError: can only concatenate list (not "int") to list
```

We can do the same thing for subtraction, multiplication, etc.:

```
In [ ]: print("Subtraction: \t" + str(myarray - 2))
        print("Multiplication:\t" + str(myarray * 10))
        print("Squared: \t" + str(myarray ** 2))
        print("Square root: \t" + str(np.sqrt(myarray)))
        print("Exponential: \t" + str(np.exp(myarray)))
```

0.3.2 Working with multiple arrays

We can also easily do these operations for multiple arrays. For example, let's say we want to add the corresponding elements of two lists together. Here's how we'd do it with regular lists:

```
In [ ]: list_a = [1, 2, 3, 4, 5]
        list_b = [6, 7, 8, 9, 10]
        list_c = [list_a[i] + list_b[i] for i in range(len(list_a))]
        list_c
```

With NumPy arrays, we just have to add the arrays together:

```
In [ ]: array_a = np.array(list_a) # equivalent to np.array([1, 2, 3, 4, 5])
        array_b = np.array(list_b) # equivalent to np.array([6, 7, 8, 9, 10])
        array_c = array_a + array_b
        array_c
```

Note: make sure when adding arrays that you are actually working with arrays, because if you try to add two lists, you will not get an error. Instead, the lists will be concatenated:

```
In [ ]: list_a + list_b
```

Just as when we are working with a single array, we can add, subtract, divide, multiply, etc. several arrays together:

```
In [ ]: print("Subtraction: \t" + str(array_a - array_b))
        print("Multiplication:\t" + str(array_a * array_b))
        print("Exponent: \t" + str(array_a ** array_b))
        print("Division: \t" + str(array_a / array_b))
```

0.4 Creating and modifying arrays

One thing that you can do with lists that you cannot do with NumPy arrays is adding and removing elements. For example, I can create a list and then add elements to it with `append`:

```
In [ ]: mylist = []
        mylist.append(7)
        mylist.append(2)
        mylist
```

However, you cannot do this with NumPy arrays. If you tried to run the following code, for example:

```
myarray = np.array([])
myarray.append(7)
```

You'd get an error like this:

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-25-0017a7f2667c> in <module>()
      1 myarray = np.array([])
----> 2 myarray.append(7)

AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

To create a NumPy array, you must create an array with the correct shape from the beginning. However, the array doesn't have to have all the correct values from the very beginning: these you can fill in later.

There are a few ways to create a new array with a particular size:

- `np.empty(size)` – creates an empty array of size `size`
- `np.zeros(size)` – creates an array of size `size` and sets all the elements to zero
- `np.ones(size)` – creates an array of size `size` and sets all the elements to one

So the way that we would create an array like the list above is:

```
In [ ]: myarray = np.empty(2) # create an array of size 2
        myarray[0] = 7
        myarray[1] = 2
        myarray
```

Another very useful function for creating arrays is np.arange, which will create an array containing a sequence of numbers (it is very similar to the built-in range or xrange functions in Python).

Here are a few examples of using `np.arange`. Try playing around with them and make sure you understand how it works:

```
In [ ]: # create an array of numbers from 0 to 3
        np.arange(3)
```

```
In [ ]: # create an array of numbers from 1 to 5
        np.arange(1, 5)
```

```
In [ ]: # create an array of every third number between 2 and 10
        np.arange(2, 10, 3)
```

```
In [ ]: # create an array of numbers between 0.1 and 1.1 spaced by 0.1
        np.arange(0.1, 1.1, 0.1)
```

0.5 "Vectorized" computations

Another very useful thing about NumPy is that it comes with many so-called "vectorized" operations. A vectorized operation (or computation) works across the entire array. For example, let's say we want to add together all the numbers in a list. In regular Python, we might do it like this:

```
In [ ]: mylist = [3, 6, 1, 10, 22]
        total = 0
        for number in mylist:
            total += number
        total
```

Using NumPy arrays, we can just use the `np.sum` function:

```
In [ ]: # you can also just do np.sum(mylist) -- it converts it to an
        # array for you!
        myarray = np.array(mylist)
        np.sum(myarray)
```

There are many other vectorized computations that you can do on NumPy arrays, including multiplication (np.prod), mean (np.mean), and variance (np.var). They all act essentially the same way as np.sum – give the function an array, and it computes the relevant function across all the elements in the array.

0.5.1 Exercise: Euclidean distance (2 points)

Recall that the Euclidean distance d is given by the following equation:

$$d(a,b) = \sqrt{\sum_{i=1}^{N}(a_i - b_i)^2}$$

In NumPy, this is a fairly simple computation because we can rely on array computations and the `np.sum` function to do all the heavy lifting for us.

Complete the function euclidean_distance below to compute $d(a,b)$, as given by the equation above. Note that you can compute the square root using np.sqrt.

```
In [ ]: def euclidean_distance(a, b):
            """Computes the Euclidean distance between a and b.

            Hint: your solution can be done in a single line of code!

            Parameters
            ----------
            a, b : numpy arrays or scalars with the same size
```

```
    Returns
    -------
    the Euclidean distance between a and b

    """
    ### BEGIN SOLUTION
    return np.sqrt(np.sum((a - b) ** 2))
    ### END SOLUTION
```

Remember that you need to execute the cell above (with your definition of euclidean_distance), and then run the cell below to check your answer. If you make changes to the cell with your answer, you will need to first re-run that cell, and then re-run the test cell to check your answer again.

```
In [ ]: # add your own test cases in this cell!
```

```
In [ ]: from nose.tools import assert_equal, assert_raises

        # check euclidean distance of size 3 integer arrays
        a = np.array([1, 2, 3])
        b = np.array([4, 5, 6])
        assert_equal(euclidean_distance(a, b), 5.196152422706632)

        # check euclidean distance of size 4 float arrays
        x = np.array([3.6, 7., 203., 3.])
        y = np.array([6., 20.2, 1., 2.])
        assert_equal(euclidean_distance(x, y), 202.44752406487959)

        # check euclidean distance of scalars
        assert_equal(euclidean_distance(1, 0.5), 0.5)

        # check that an error is thrown if the arrays are different sizes
        a = np.array([1, 2, 3])
        b = np.array([4, 5])
        assert_raises(ValueError, euclidean_distance, a, b)
        assert_raises(ValueError, euclidean_distance, b, a)

        print("Success!")
```

0.6 Creating multidimensional arrays

Previously, we saw that functions like `np.zeros` or `np.ones` could be used to create a 1-D array. We can also use them to create N-D arrays. Rather than passing an integer as the first argument, we pass a list or tuple with the <u>shape</u> of the array that we want. For example, to create a 3×4 array of zeros:

```
In [ ]: arr = np.zeros((3, 4))
        arr
```

The <u>shape</u> of the array is a very important concept. You can always get the shape of an array by accessing its shape attribute:

```
In [ ]: arr.shape
```

Note that for 1-D arrays, the shape returned by the `shape` attribute is still a tuple, even though it only has a length of one:

```
In [ ]: np.zeros(3).shape
```

This also means that we can create 1-D arrays by passing a length one tuple. Thus, the following two arrays are identical:

```
In [ ]: np.zeros((3,))
```

```
In [ ]: np.zeros(3)
```

There is a warning that goes with this, however: be careful to always use tuples to specify the shape when you are creating multidimensional arrays. For example, to create an array of zeros with shape (3, 4), we must use np.zeros((3, 4)). The following will not work:

```
np.zeros(3, 4)
```

It will give an error like this:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-06beb765944a> in <module>()
----> 1 np.zeros(3, 4)

TypeError: data type not understood
```

This is because the second argument to `np.zeros` is the data type, so numpy thinks you are trying to create an array of zeros with shape `(3,)` and datatype `4`. It (understandably) doesn't know what you mean by a datatype of `4`, and so throws an error.

Another important concept is the size of the array – in other words, how many elements are in it. This is equivalent to the length of the array, for 1-D arrays, but not for multidimensional arrays. You can also see the total size of the array with the size attribute:

```
In [ ]: arr = np.zeros((3, 4))
        arr.size
```

We can also create arrays and then reshape them into any shape, provided the new array has the same size as the old array:

```
In [ ]: arr = np.arange(32).reshape((8, 4))
        arr
```

0.7 Accessing and modifying multidimensional array elements

To access or set individual elements of the array, we can index with a sequence of numbers:

```
In [ ]: # set the 3rd element in the 1st row to 0
        arr[0, 2] = 0
        arr
```

We can also access the element on it's own, without having the equals sign and the stuff to the right of it:

```
In [ ]: arr[0, 2]
```

We frequently will want to access ranges of elements. In NumPy, the first dimension (or axis) corresponds to the rows of the array, and the second axis corresponds to the columns. For example, to look at the first row of the array:

```
In [ ]: # the first row
        arr[0]
```

To look at columns, we use the following syntax:

```
In [ ]: # the second column
        arr[:, 1]
```

The colon in the first position essentially means "select from every row". So, we can interpret `arr[:, 1]` as meaning "take the second element of every row", or simply "take the second column".

Using this syntax, we can select whole regions of an array. For example:

```
In [ ]: # select a rectangular region from the array
        arr[2:5, 1:3]
```

Note: be careful about setting modifying an array if what you really want is a copy of an array. Remember that in Python, variables are really just pointers to objects.

For example, if I want to create a second array that mutliples every other value in `arr` by two, the following code will work but will have unexpected consequences:

```
In [ ]: arr = np.arange(10)
        arr2 = arr
        arr2[::2] = arr2[::2] * 2
        print("arr:  " + str(arr))
        print("arr2: " + str(arr2))
```

Note that `arr` and `arr2` both have the same values! This is because the line `arr2 = arr` doesn't actually copy the array: it just makes another pointer to the same object. To truly copy the array, we need to use the `.copy()` method:

```
In [ ]: arr = np.arange(10)
        arr2 = arr.copy()
        arr2[::2] = arr2[::2] * 2
        print("arr:  " + str(arr))
        print("arr2: " + str(arr2))
```

0.7.1 Exercise: Border (2 points)

Write a function to create a 2D array of arbitrary shape. This array should have all zero values, except for the elements around the border (i.e., the first and last rows, and the first and last columns), which should have a value of one.

```
In [ ]: def border(n, m):
            """Creates an array with shape (n, m) that is all zeros
            except for the border (i.e., the first and last rows and
            columns), which should be filled with ones.

            Hint: you should be able to do this in three lines
            (including the return statement)

            Parameters
            ----------
            n, m: int
                Number of rows and number of columns

            Returns
            -------
            numpy array with shape (n, m)
```

```
    """
    ### BEGIN SOLUTION
    arr = np.ones((n, m))
    arr[1:-1, 1:-1] = 0
    return arr
    ### END SOLUTION
```

In [ ]:
```
# add your own test cases in this cell!
```

In [ ]:
```
from numpy.testing import assert_array_equal
from nose.tools import assert_equal

# check a few small examples explicitly
assert_array_equal(border(1, 1), [[1]])
assert_array_equal(border(2, 2), [[1, 1], [1, 1]])
assert_array_equal(border(3, 3), [[1, 1, 1], [1, 0, 1], [1, 1, 1]])
assert_array_equal(border(3, 4), [[1, 1, 1, 1], [1, 0, 0, 1], [1, 1, 1, 1]])

# check a few large and random examples
for i in range(10):
    n, m = np.random.randint(2, 1000, 2)
    result = border(n, m)

    # check dtype and array shape
    assert_equal(result.dtype, np.float)
    assert_equal(result.shape, (n, m))

    # check the borders
    assert (result[0] == 1).all()
    assert (result[-1] == 1).all()
    assert (result[:, 0] == 1).all()
    assert (result[:, -1] == 1).all()

    # check that everything else is zero
    assert np.sum(result) == (2*n + 2*m - 4)

print("Success!")
```