ESCI 386 – Scientific Programming, Analysis and Visualization with Python

Lesson 3 – NumPy Arrays

About NumPy

- NumPy stands for *Numerical Python*.
- Pronunciation?
  - Num-pea?
  - Num-pie?
- Has many of the same functions as does the Math module, but Numpy is more extensive.
  - I almost exclusively use NumPy over Math

Importing NumPy

- NumPy is commonly imported and aliased as `np`.

```
import numpy as np
```

NumPy Arrays

Python does not have a built-in array data type.

It does not have a module called ARRAY that has objects called arrays.

These arrays behave essentially like lists that are forced to all have the same data type for their elements.

NumPy has array objects that behave more like Fortran or IDL arrays.

Python Arrays vs. NumPy Arrays

- **Python does not have a built-in array data type.**
- **Python does have a module called ARRAY that has objects called arrays.**
  - **These objects behave essentially like lists that are forced to all have the same data type for their elements.**
- **NumPy has array objects that behave more like Fortran or IDL arrays.**

Creating a 1-D Array

```
>>> a = np.array([3, -5, 8])
>>> print(a)
[ 3 -5  8]
```

Creating a 2-D Array

```
>>> a = np.array([[3, -5, 8], [-7, 6, 9]])
>>> print(a)
[[ 3 -5  8]
 [-7  6  9]]
```

Array Indexing

- Array indices begin at 0
  - Different from Fortran

- For 2-D arrays the first index is the row, and the second index is the column

Ranges of Indices

- A colon is used to specify a range if indices.
- Form is *m*:*n*
- This refers to indices from *m* to *n*−1

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:6]
array([2, 3, 4, 5])
```

Ranges of Indices (cont.)

- All elements from the beginning of the array to element to *n* can be accessed by `:`*n*`+1`

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[:4]
array([0, 1, 2, 3])
```

Ranges of Indices (cont.)

- All elements from the index *n* to the end of the array can be accessed by `n:`

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```

Using Ranges with 2-D Arrays

```
>>> a
array([[ 3, -5,  8],
       [-7,  6,  9],
       [ 2,  5, -3]])
>>> a[1,:]
array([-7,  6,  9])
>>> a[:,1]
array([-5,  6,  5])
>>> a[1:,0]
array([-7,  2])
```

Creating Zeroed Array

```
>>> a = np.zeros((5,3), dtype = np.float64)
>>> a
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Creating Single Valued Array

```
>>> a = np.ones((5,3), dtype = np.float64)
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = a*4
>>> b
array([[ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.]])
```

Creating 'Empty' Array

```
>>> a = np.empty((5,3),dtype = np.float64)
>>> a
array([[  3.38211339e-306,   1.27873747e+294,   4.71524470e-309],
       [  5.83307315e-302,   1.95592910e+289,   4.84562028e-309],
       [  3.41149048e-309,   1.63321437e-301,   4.95081600e+173],
       [  4.11141936e-282,   3.91957011e+202,   6.73606881e-310],
       [  9.16700530e-278,   1.76125764e-312,   1.75700645e-152]])
```

Not very empty!

- NOTE: An empty array may <u>not be really empty</u>!
- Don't assume initial values are zero or nil.

Creating Arrays with Same Shape and Type as Existing Array

```
>>> a
array([[ 3.4, -2.7,  9. ],
       [-1.8,  3.8, -2.4]])
>>> b = np.zeros_like(a)
>>> b
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> b = np.ones_like(a)
>>> b
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.empty_like(a)
>>> b
array([[ 3.4,  2.7,  9. ],
       [ 1.8,  3.8,  2.4]])
```

Creating Arrays with `numpy.arange()`

```
>>> a = np.arange(0,10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = np.arange(0,10.0)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a = np.arange(0,10,0.5)
>>> a
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
>>> a = np.arange(-5,5)
>>> a
array([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> a = np.arange(10, 0, -1)
>>> a
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

Creating Arrays with `numpy.linspace()`

```
>>> a = np.linspace(-5, 5,20)
>>> a
array([-5.        , -4.47368421, -3.94736842, -3.42105263, -2.89473684,
       -2.36842105, -1.84210526, -1.31578947, -0.78947368, -0.26315789,
        0.26315789,  0.78947368,  1.31578947,  1.84210526,  2.36842105,
        2.89473684,  3.42105263,  3.94736842,  4.47368421,  5.        ])
>>> a = np.linspace(5, -5, 20)
>>> a
array([ 5.        ,  4.47368421,  3.94736842,  3.42105263,  2.89473684,
        2.36842105,  1.84210526,  1.31578947,  0.78947368,  0.26315789,
       -0.26315789, -0.78947368, -1.31578947, -1.84210526, -2.36842105,
       -2.89473684, -3.42105263, -3.94736842, -4.47368421, -5.        ])
```

Creating Arrays with `numpy.logspace()`

```
>>> a = np.logspace(0,4,10)
>>> a
array([  1.00000000e+00,   2.78255940e+00,   7.74263683e+00,
         2.15443469e+01,   5.99484250e+01,   1.66810054e+02,
         4.64158883e+02,   1.29154967e+03,   3.59381366e+03,
         1.00000000e+04])
```

Copying Arrays

- NumPy has its own function for making a deep copy of an array.
  - `np.copy(`*`array`*`)`

The `where` Function

- Used to find indices of an array where elements meet certain condition

```
>>> a
array([  3,  -5, -10,   8,   0,  -5])
>>> result = np.where(a < 0)
>>> result
(array([1, 2, 5], dtype=int64),)
>>> a[result]
array([ -5, -10,  -5])
>>> a[result] = 0
>>> a
array([3, 0, 0, 8, 0, 0])
```

Saving NumPy Arrays

- NumPy provides its own functions to read and write arrays to binary files. This is accomplished with either:
  - `np.save()` function, which writes a single array to a NumPy .npy file.
  - `np.savez()` function, which archives several arrays into a NumPy .npz file.

Example

```
import numpy as np
a = np.arange(0, 100)*0.5
b = np.arange(-100, 0)*0.5
np.save('a-file', a)
np.save('b-file', b)
np.savez('ab-file', a=a, b=b)
```

- Creates three files:
  - a-file.npy which contains the values for a
  - b-file.npy which contains the values for b
  - ab-file.npz which is an archive file containing both the a and b values

Loading .npy Files

- To retrieve the values from the .npy files we use the `np.load()` function

```
a = np.load('a-file.npy')
b = np.load('b-file.npy')
```

Loading .npz Files

- To retrieve the values from the .npz files we also use the `np.load()` function to load all the data into a dictionary that contains the archived arrays.

```
z = np.load('ab-file.npz')
a = z['a']
b = z['b']
```

Loading .npz Files

- To find the names of the arrays used in the dictionary, use the `files` attribute of the dictionary

```
>>> z.files
['a', 'b']
```