NUMERICAL METHODS

Python Crash Course

Part 6

OPEN A JUPYTER NOTEBOOK AND TRY THESE DURING CLASS

YOUR NEW FAVOURITE PACKAGE: NUMPY

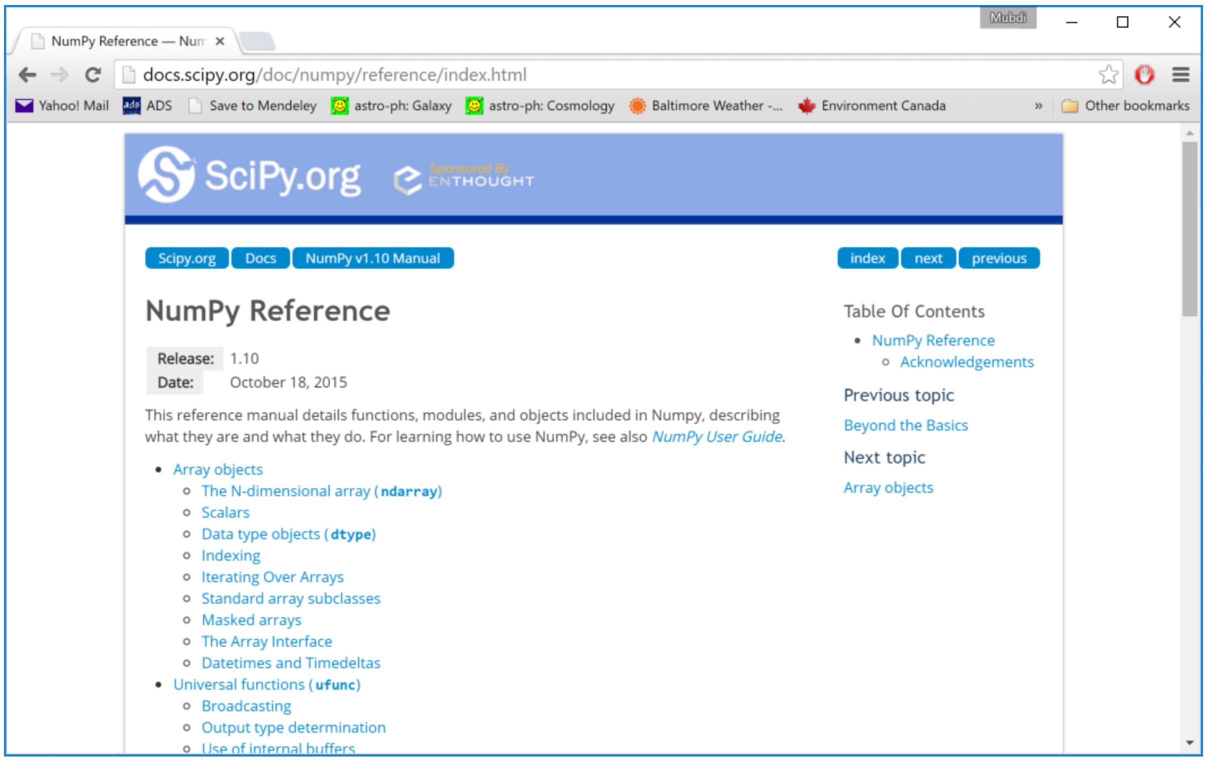
Numpy provides python with its numerical muscle. **This is your go-to package.** The package is written in C and made to deal with N-dimensional arrays, all basic mathematical operations, linear algebra operations, *et cetera*. **We will not be going through all of the power of this module.**

Importing the Numpy module:

```
import numpy as np
```

You will be using numpy for basically all of your future python work. Make sure you import it at the beginning of any/every script and when you first open ipython or a Jupyter notebook.

THE DOCUMENTATION



Your reference to all Numpy goodness:
http://docs.scipy.org/doc/numpy/reference/index.html

PRO TIP: Bookmark it now

NUMPY ARRAYS: YOUR BEST FRIEND

Numpy arrays are the base object containing a variety of powerful methods. Making a Numpy array is easy:

```
In [1]: array1 = np.array([1, 2, 3]) # One-liner
```

PRO TIP:

Be careful it is

```
np.array([1, 2, 3])          # RIGHT
```

not

```
np.array[1, 2, 3]            # WRONG
```

NUMPY ARRAYS: YOUR BEST FRIEND

Numpy arrays are the base object containing a variety of powerful methods. Making a Numpy array is easy:

```
In [1]: array1 = np.array([1, 2, 3]) # One-liner
In [2]: list1 = [1.0, 2.0, 3.4]
In [3]: array2 = np.array(list1) # List -> NP Array
```

All data in a Numpy array *must* be of a single data type (dtype). Numpy has a large number of possible data types:

```
np.str # string
np.bool # boolean (i.e., True|False)
np.int # integer
np.float # floating point
np.complex # complex (i.e., 1+1j)
```

NUMPY ARRAYS: YOUR BEST FRIEND

Numpy arrays are the base object containing a variety of powerful methods. Making a Numpy array is easy:

```
In [1]: array1 = np.array([1,
In [2]: list1 = [1.0, 2.0, 3.
In [3]: array2 = np.array(lis
```

All data in a Numpy array *must* be
Numpy has a large number of poss

```
np.str # string
np.bool # boolean (i.e., True
np.int # integer
np.float # floating point
np.complex # complex (i.e., 1+1j)
```

PRO TIP:

Any of the data type objects have an associated function (with an underscore) to convert a Python array of one sort into another:

```
arr1 = np.array([1, 2])
arr2 = np.float_(arr1)
```

NUMPY ARRAYS: MULTIDIMENSIONALITY

Numpy arrays can be N-dimensional, which is of particular use with tables of data (i.e. 2-D).

```
# Creating a 3x2 Array:
array1 = np.array([[1, 2], [3, 4], [5, 6]])
array1[:,1] # Results in array([2, 4, 6])
array1.shape # Results in (3, 2)
array1.size # Results in 6

array1.flatten() # Returns 1-dimensional array
array1.reshape((6,1)) # Returns a 2-D array
```

PRO TIP:

There is a difference between a 1-dimensional array and a 2-dimensional array with one of the axes having length 1.

NUMPY ARRAYS: HOW DO YOU MAKE THEM?

Making them manually:

```
# Creating a 2x2 Array:
array1 = np.array([[1, 2], [3, 4]]) # All ints
array1 = np.array([[1.0, 2], [3, 4]]) # All floats
```

Or using special functions:

```
# Creating arrays filled sequentially
array2 = np.arange(10) # [0, 1, 2, … 9]

# Creating arrays filled with specific values:
array3 = np.ones(10) # 1-D array filled with 1s
array4 = np.zeros((3,5)) # 2-D array filled with 0s
array5 = np.identity(5) # 5x5 identity matrix
```

NUMPY ARRAYS: WHAT CAN YOU DO WITH THEM?

Every Numpy array has lots of built in functionality:

```
# Assuming array1 is a (3,3) numpy array:
array = np.arange(9).reshape((3,3))

# Basic Parameters:
array1.min(), array1.max()

# Taken over one whole array:
array1.mean(), array1.sum(), array1.prod()

# Taken in one dimension along first axis:
array1.mean(axis=0), array1.sum(axis=0)

# Rearrange data as a copy:
array1.transpose(), array1.T, array1.swapaxes(0, 1)
```

All this functionality is also provided in functions that take the array as the first argument: `np.mean(array1)`

NUMPY ARRAYS: RESHAPING REDUX

When reshaping a Numpy array, you can leave one of the axes lengths to be -1, which causes python to determine its length.

```
array1 = np.arange(90).reshape((-1, 10))
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89

NUMPY ARRAYS: RESHAPING REDUX

```
array1.shape == (9, 10)
# Returns True
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89

NUMPY ARRAYS: INDEXING REDUX

```
array1[4,6] = 0
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	0	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89

NUMPY ARRAYS: INDEXING REDUX

```
array1[5,:] = [1,1,1,1,1,1,1,1,1,1]
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	0	47	48	49
1	1	1	1	1	1	1	1	1	1
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89

NUMPY ARRAYS: INDEXING REDUX

```
array1[5,1:8] = [2,2,2,2,2,2,2]
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	0	47	48	49
1	2	2	2	2	2	2	2	1	1
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89

NUMPY ARRAYS: INDEXING REDUX

```
# -1 means "last"
array1[5:-1,1:3] = [[3,3],[3,3],[3,3]]
```

	0	1	2	3	4	5	6	7	8	9
	10	11	12	13	14	15	16	17	18	19
	20	21	22	23	24	25	26	27	28	29
	30	31	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	0	47	48	49
5	1	3	3	2	2	2	2	2	1	1
	60	3	3	63	64	65	66	67	68	69
	70	3	3	73	74	75	76	77	78	79
	80	81	82	83	84	85	86	87	88	89
		1	2							

NUMPY ARRAYS: INDEXING REDUX

```
array1[(5, 7), (6, 8)] = [4,4]
```

Non-sequential indexing!

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	0	47	48	49
1	3	3	2	2	2	4	2	1	1
60	3	3	63	64	65	66	67	68	69
70	3	3	73	74	75	76	77	4	79
80	81	82	83	84	85	86	87	88	89

NUMPY ARRAYS: CAUTIONARY TALE

Numpy arrays are generally passed by reference (to minimize space used in memory):

```
In [1]: array1 = np.array([1, 2, 3])
In [2]: array2 = array1[0:2] # Now [1, 2]
In [3]: array1[0] = 5 # Now [5, 2, 3]
In [4]: array2 == np.array([5, 2]) # value edited
```

To ensure that values are independent, use the copy function:

```
In [5]: array2 = np.copy(array1[0:2]) # or
In [6]: array2 = np.array(array1[0:2], copy=True)
```

READING IN TEXT (ASCII) FILES

There are multiple ways of reading in files, but we'll concentrate on the 'loadtxt' function:

```
# Works with a perfectly formatted file
array1 = np.loadtxt('../files/dat_files/mydata.dat')
```

There are lots of options on this function, so check the docs, but some of the most used:

```
array2 = np.loadtxt(filename, dtype=dtype, comments='#',
delimiter=',', skiprows=5,
usecols=(0, 1, 2))
# This skips all comments (designated with a #) and
# the first 5 rows. It then reads in columns 0, 1,
# and 2, delimited by a comma
```

READING IN TEXT (ASCII) FILES (CONT…)

When in doubt about arguments and what form they should be, check the docs:

```
np.loadtxt?
```

Loadtxt can read in gzipped (.gz) and Bzip2 (.bz2) files without them being unzipped.

MATHEMATICAL OPERATIONS

Mathematical operations proceed *element-wise*:

```
# These act on each element
array1 + 5, array1 * 5, array1**2

# These act element to element (if the same size)
array2 = np.copy(array1)
array1 + array2, array1*array2
```

And our favourite mathematical operations:

```
# These act on either numpy arrays, floats, or ints
np.log10(array1), np.exp(array1), np.sin(array1),
np.cosh(array1)
```

Numpy has some useful constants defined, like: `np.pi, np.e`

MATRIX MATH

For 2-D (and higher) matrices, you can do standard matrix math:

```
arr1 = np.array([[0,1],[2,3]])
arr2 = np.array([[4,5],[6,7]])

# Doing standard matrix math:
np.dot(arr1, arr2)
np.cross(arr1, arr2)

# More complex operations
np.linalg.eig(arr1)  # Eigenvalues and eigenvectors
np.linalg.det(arr1)

# Even calculating inverses:
np.linalg.inv(arr1)
```

SEARCHING ARRAYS

You can search for specific values in an array using the where command:

```
arr1 = np.arange(6).reshape((2,3))
# arr1 = array([[0, 1, 2],
#                [3, 4, 5]])

np.where(arr1 > 1)
# Return row/col tuple
# (array([0, 1, 1, 1]), array([2, 0, 1, 2]))
# which basically gives which elements of arr1 > 1, i.e.
# array([0, 2]) which is 2
```

SEARCHING ARRAYS

You can search for specific values in an array using the where command:

```
arr1 = np.arange(6).reshape((2,3))
inds1 = np.where(arr1 > 1)  # Return rows and columns
inds2 = np.where((arr1 > 2) & (arr1 < 6)) # Logical AND

# Can use multiple arrays in a single command:
# Arrays must be the same size and shape
arr2 = np.copy(arr1) + 1
inds2 = np.where((arr1 > 4) & (arr2 < 10))
```

The indices given are a tuple with length equalling the number of dimensions of the array. The tuple contains numpy arrays of the index of the matching value in each dimension:

```
inds1 == (array([0, 4, … , 10])) # One dimension
inds2 == (array([0, … , 10]), array([3, … , 5])) #2D
```

If there’s no values that match your conditions, where will produce a tuple of empty numpy arrays.

VECTORIZING FUNCTIONS

Sometimes, you'll want to make complex functions that don't necessarily automatically work with numpy arrays:

```
def funct1(val):
    if val > 3: # Doesn't work with array
         x = 2
    else:
        x = 5
    return x
```

The vectorize function makes functions like this work for arrays:

```
z = [2,4,6]
funct1(z)  # Fails
vfunct1 = np.vectorize(funct1)
vfunct1(z) # Now works!
```

While this is fine to do for functions you don't need high performance on, it is slow(ish). Consider writing the function better for speed.

SAVING YOUR OUTPUT

For individual numpy arrays, there are some quick and dirty methods to save your data:

```
# Quick and Dirty in Text:
x = np.arange(100).reshape((25,4))
np.savetxt('test.dat', x)
```

Numpy also has some proprietary formats (.npy, .npz) that allow for quick reading of data:

```
# Saving a single array:
np.save('test.npy', arr1)

# Saving multiple arrays:
x2 = np.arange(20).reshape((5,4))
np.savez('test', a1=x, a2=x2)   # .npz suffix added
```

Output file extensions are based on how many arrays you have in the save file: .npy is for a single array and .npz is for multiple

LOADING SAVED OUTPUT

To load a single numpy array (.npy file):

```
arr1 = np.load('test.npy')
```

To load a multiple numpy arrays (.npz file):

```
alldata = np.load('test.npz')

# All Data Object is dictionary-like:
var1 = alldata['a1']
var2 = alldata['a2']
```

LAMBDA FUNCTIONS

Sometimes you want to define a simple function without the full function syntax. **Lambda functions** exist for this exact reason:

```
# Defining the Function:
funct1 = lambda x: x**2 # Returns the square of x

# Using the Function:
tmpvar1 = funct1(5)

# Can use multiple variables:
funct2 = lambda x,y: x + y

# Using the Function:
tmpvar2 = funct2(5, 6)
```

EXERCISE TIME!