



ATMS 305 WEEK 6: USING NUMPY

Lecture 1: Practical uses of
numpy

SCIENTIFIC PYTHON?

Extra features required:

- fast, multidimensional arrays
- libraries of reliable, tested scientific functions
- plotting tools

NumPy is at the core of nearly every scientific Python application or module since it provides a fast N-d array datatype that can be manipulated in a vectorized form.

ARRAYS — NUMERICAL PYTHON (NUMPY)

Lists ok for storing small amounts of one-dimensional data

```
>>> a = [1,3,5,7,9]
>>> print(a[2:4])
[5, 7]
>>> b = [[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
>>> print(b[0])
[1, 3, 5, 7, 9]
>>> print(b[1][2:4])
[6, 8]
```

```
>>> a = [1,3,5,7,9]
>>> b = [3,5,6,7,9]
>>> c = a + b
>>> print c
[1, 3, 5, 7, 9, 3, 5, 6, 7, 9]
```

- But, can't use directly with arithmetical operators (+, -, *, /, ...)
- Need efficient arrays with arithmetic and better multidimensional tools
- **Numpy**

```
>>> import numpy as np
```
- Similar to lists, but much more capable, except fixed size

NUMPY — N-DIMENSIONAL ARRAY MANIPULATIONS

The fundamental library needed for scientific computing with Python is called NumPy. This Open Source library contains:

- a powerful N-dimensional array object
- advanced array slicing methods (to select array elements)
- convenient array reshaping methods

and it even contains 3 libraries with numerical routines:

- basic linear algebra functions
- basic Fourier transforms
- sophisticated random number capabilities

NumPy can be extended with C-code for functions where performance is highly time critical. In addition, tools are provided for integrating existing Fortran code. NumPy is a hybrid of the older NumArray and Numeric packages, and is meant to replace them both.

NUMPY — CREATING ARRAYS

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files

NUMPY — CREATING VECTORS

From lists

- `np.array`

```
# as vectors from lists
>>> a = np.array([1,3,5,7,9])
>>> b = np.array([3,5,6,7,9])
>>> c = a + b
>>> print c
[4, 8, 11, 14, 18]

>>> type(c)
(<type 'np.ndarray'>)

>>> c.shape
(5,)
```

NUMPY — CREATING MATRICES

```
>>> l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]] # create a list
```

```
>>> a = np.array(l) # convert a list to an array
```

```
>>> print(a)
```

```
[[1 2 3]
```

```
 [3 6 9]
```

```
 [2 4 6]]
```

```
>>> a.shape
```

```
(3, 3)
```

```
>>> print(a.dtype) # get type of an array
```

```
int64
```

```
# or directly as matrix
```

```
>>> M = array([[1, 2], [3, 4]])
```

```
>>> M.shape
```

```
(2, 2)
```

```
>>> M.dtype
```

```
dtype('int64')
```

```
#only one type
```

```
>>> M[0,0] = "hello"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: invalid literal for long() with base 10: 'hello'
```

```
>>> M = np.array([[1, 2], [3, 4]], dtype=complex)
```

```
>>> M
```

```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

NUMPY — MATRICES USE

```
>>> print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]
>>> print(a[0]) # this is just like a list of lists
[1 2 3]
>>> print(a[1, 2]) # arrays can be given comma separated indices
9
>>> print(a[1, 1:3]) # and slices
[6 9]
>>> print(a[:,1])
[2 6 4]
>>> a[1, 2] = 7
>>> print(a)
[[1 2 3]
 [3 6 7]
 [2 4 6]]
>>> a[:, 0] = [0, 9, 8]
>>> print(a)
[[0 2 3]
 [9 6 7]
 [8 4 6]]
```


NUMPY — CREATING ARRAYS

Generation functions

```
>>> x = arange(0, 10, 1) # arguments: start, stop, step
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> np.linspace(0, 10, 25)
array([ 0.          ,  0.41666667,  0.83333333,  1.25          ,
        1.66666667,  2.08333333,  2.5          ,  2.91666667,
        3.33333333,  3.75          ,  4.16666667,  4.58333333,
        5.          ,  5.41666667,  5.83333333,  6.25          ,
        6.66666667,  7.08333333,  7.5          ,  7.91666667,
        8.33333333,  8.75          ,  9.16666667,  9.58333333, 10.          ])

>>> np.logspace(0, 10, 10, base=np.e)
array([ 1.00000000e+00,  3.03773178e+00,  9.22781435e+00,
        2.80316249e+01,  8.51525577e+01,  2.58670631e+02,
        7.85771994e+02,  2.38696456e+03,  7.25095809e+03,
        2.20264658e+04])
```

NUMPY — CREATING ARRAYS

```
# a diagonal matrix
>>> np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])

>>> b = np.zeros(5)
>>> print(b)
[ 0.  0.  0.  0.  0.]
>>> b.dtype
dtype('float64')
>>> n = 1000
>>> my_int_array = np.zeros(n, dtype=np.int)
>>> my_int_array.dtype
dtype('int32')

>>> c = np.ones((3,3))
>>> c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

NUMPY — ARRAY CREATION AND USE

```
>>> d = np.arange(5) # just like range()
>>> print(d)
[0 1 2 3 4]

>>> d[1] = 9.7
>>> print(d) # arrays keep their type even if elements changed
[0 9 2 3 4]

>>> print(d*0.4) # operations create a new array, with new type
[ 0.   3.6  0.8  1.2  1.6]

>>> d = np.arange(5, dtype=np.float)
>>> print(d)
[ 0.  1.  2.  3.  4.]

>>> np.arange(3, 7, 0.5) # arbitrary start, stop and step
array([ 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

NUMPY — ARRAY CREATION AND USE

```
>>> x, y = np.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
>>> x
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
# random data
>>> np.random.rand(5,5)
array([[ 0.51531133,  0.74085206,  0.99570623,  0.97064334,  0.5819413 ],
       [ 0.2105685 ,  0.86289893,  0.13404438,  0.77967281,  0.78480563],
       [ 0.62687607,  0.51112285,  0.18374991,  0.2582663 ,  0.58475672],
       [ 0.72768256,  0.08885194,  0.69519174,  0.16049876,  0.34557215],
       [ 0.93724333,  0.17407127,  0.1237831 ,  0.96840203,  0.52790012]])
```

NUMPY — CREATING ARRAYS

File I/O

```
>>> os.system('head DeBilt.txt')
"Stn", "Datum", "Tg", "qTg", "Tn", "qTn", "Tx", "qTx"
001, 19010101,    -49, 00,    -68, 00,    -22, 40

0 >>> np.savetxt('datasaved.txt', data)
0 >>> os.system('head datasaved.txt')
0 1.0000000000000000e+00 1.9010101000000000e+07 -4.9000000000000000e+01
0 0.0000000000000000e+00 -6.8000000000000000e+01 0.0000000000000000e+00 -
0 2.2000000000000000e+01 4.0000000000000000e+01
0 1.0000000000000000e+00 1.9010102000000000e+07 -2.1000000000000000e+01
0 0.0000000000000000e+00 -3.6000000000000000e+01 3.0000000000000000e+01 -
0 1.3000000000000000e+01 3.0000000000000000e+01
0 1.0000000000000000e+00 1.9010103000000000e+07 -2.8000000000000000e+01
0 0.0000000000000000e+00 -7.9000000000000000e+01 3.0000000000000000e+01 -
> 5.0000000000000000e+00 2.0000000000000000e+01
>
(25568, 8)
```

NUMPY — CREATING ARRAYS

```
>>> M = np.random.rand(3,3)
>>> M
array([[ 0.84188778,  0.70928643,  0.87321035],
       [ 0.81885553,  0.92208501,  0.873464  ],
       [ 0.27111984,  0.82213106,  0.55987325]])
>>>
>>> np.save('saved-matrix.npy', M)
>>> np.load('saved-matrix.npy')
array([[ 0.84188778,  0.70928643,  0.87321035],
       [ 0.81885553,  0.92208501,  0.873464  ],
       [ 0.27111984,  0.82213106,  0.55987325]])
>>>
>>> os.system('head saved-matrix.npy')
NUMPYF{'descr': '<f8', 'fortran_order': False, 'shape': (3, 3), }
İ<
f%ðê?sy²æ?$/÷òvñê?ù4ê?%dn,í?Ã[Äjóë?Ä,zÑ?Ç
îânê?ó7L{êá?0
>>>
```

NUMPY - NDARRAY

NumPy's main object is the homogeneous multidimensional array called `ndarray`.

- This is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. Typical examples of multidimensional arrays include vectors, matrices, images and spreadsheets.
- Dimensions usually called axes, number of axes is the rank

`[7, 5, -1]` An array of rank 1 i.e. It has 1 axis of length 3

`[[1.5, 0.2, -3.7] ,` An array of rank 2 i.e. It has 2 axes, the first
`[0.1, 1.7, 2.9]]` length 3, the second of length 3 (a matrix
with 2 rows and 3 columns

NUMPY — NDARRAY ATTRIBUTES

ndarray.ndim

- the number of axes (dimensions) of the array i.e. the rank.

ndarray.shape

- the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the rank, or number of dimensions, ndim.

ndarray.size

- the total number of elements of the array, equal to the product of the elements of shape.

ndarray.dtype

- an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. NumPy provides many, for example bool_, character, int_, int8, int16, int32, int64, float_, float8, float16, float32, float64, complex_, complex64, object_.

ndarray.itemsize

- the size in bytes of each element of the array. E.g. for elements of type float64, itemsize is 8 (=64/8), while complex32 has itemsize 4 (=32/8) (equivalent to ndarray.dtype.itemsize).

ndarray.data

- the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

NUMPY — ARRAY CREATION AND USE

Two ndarrays are mutable and may be views to the same memory:

```
>>> x = np.array([1,2,3,4])
>>> y = x
>>> x is y
True
>>> id(x), id(y)
(139814289111920, 139814289111920)
>>> x[0] = 9
>>> y
array([9, 2, 3, 4])

>>> x[0] = 1
>>> z = x[:]
>>> x is z
False
>>> id(x), id(z)
(139814289111920, 139814289112080)
>>> x[0] = 8
>>> z
array([8, 2, 3, 4])
```

```
>>> x = np.array([1,2,3,4])
>>> y = x.copy()
>>> x is y
False
>>> id(x), id(y)
(139814289111920, 139814289111840)
>>> x[0] = 9
>>> x
array([9, 2, 3, 4])
>>> y
array([1, 2, 3, 4])
```

NUMPY — ARRAY CREATION AND USE

```
>>> a = np.arange(4.0)
>>> b = a * 23.4
>>> c = b/(a+1)
>>> c += 10
>>> print c
[ 10.   21.7  25.6  27.55]

>>> arr = np.arange(100, 200)
>>> select = [5, 25, 50, 75, -5]
>>> print(arr[select]) # can use integer lists as indices
[105, 125, 150, 175, 195]

>>> arr = np.arange(10, 20 )
>>> div_by_3 = arr%3 == 0 # comparison produces boolean array
>>> print(div_by_3)
[ False False  True False False  True False False  True False]
>>> print(arr[div_by_3]) # can use boolean lists as indices
[12 15 18]

>>> arr = np.arange(10, 20) . reshape((2,5))
[[10 11 12 13 14]
 [15 16 17 18 19]]
```

NUMPY — ARRAY METHODS

```
>>> arr.sum()
145
>>> arr.mean()
14.5
>>> arr.std()
2.8722813232690143
>>> arr.max()
19
>>> arr.min()
10
>>> div_by_3.all()
False
>>> div_by_3.any()
True
>>> div_by_3.sum()
3
>>> div_by_3.nonzero()
(array([2, 5, 8]),)
```

NUMPY — ARRAY METHODS - SORTING

```
>>> arr = np.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
>>> arr.sort() # acts on array itself
>>> print(arr)
[ 1.2  1.8  2.3  4.5  5.5  6.7]

>>> x = np.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
>>> np.sort(x)
array([ 1.2,  1.8,  2.3,  4.5,  5.5,  6.7])

>>> print(x)
[ 4.5  2.3  6.7  1.2  1.8  5.5]

>>> s = x.argsort() #return indices as sorted
>>> s
array([3, 4, 1, 0, 5, 2])
>>> x[s]
array([ 1.2,  1.8,  2.3,  4.5,  5.5,  6.7])
>>> y[s]
array([ 6.2,  7.8,  2.3,  1.5,  8.5,  4.7])
```

NUMPY — ARRAY FUNCTIONS

Most array methods have equivalent functions

```
>>> arr.sum()
45
>>> np.sum(arr)
45
```

- Ufuncs provide many element-by-element math, trig., etc. operations
 - e.g., `add(x1, x2)`, `absolute(x)`, `log10(x)`, `sin(x)`, `logical_and(x1, x2)`
- See <http://numpy.scipy.org>

NUMPY – STATISTICS

In addition to the mean, var, and std functions, NumPy supplies several other methods for returning statistical features of arrays. The median can be found:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form `[[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...]` where x, y, z are different observables and the numbers indicate the observation times:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.          ,  0.72870505],
       [ 0.72870505,  1.          ]])
```

Here the return array `c[i,j]` gives the correlation coefficient for the *i*th and *j*th observables. Similarly, the covariance for data can be found::

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

USING ARRAYS WISELY

Array operations are implemented in C or Fortran

Optimised algorithms - i.e. fast!

Python loops (i.e. `for i in a:...`) are much slower

Prefer array operations over loops, especially when speed important

Also produces shorter code, often more readable

NUMPY — ARRAY MATHEMATICS

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

```
>>> a = np.array([[1, 2], [3, 4],
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b
array([-1.,  3.])
>>> a + b
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
```

```
>>> a * a
array([[ 1.,  4.],
       [ 9., 16.],
       [25., 36.]])
>>> b * b
array([ 1.,  9.])
>>> a * b
array([[ -1.,  6.],
       [ -3., 12.],
       [ -5., 18.]])
>>>
```