

IAP Python - Lecture 4

Andrew Farrell

MIT SIPB

January 13, 2011

NumPy, SciPy, and matplotlib are a collection of modules that together are trying to create the functionality of MATLAB in Python.

NumPy

NumPy is a library designed for working with multidimensional arrays, which are matrices generalized to arbitrary dimensions.

You might be tempted a file with `from numpy import *`, since you will be using numpy objects alot, but be aware that someone reading your code will have to understand that they need to know what context that puts them in. It would be better to use `import numpy as n` and then refer to any object form numpy with `n.foo`

Creating Arrays

To create an array, use a list of lists or the `arange` function, which is just like `range` except that it returns a 1-D array and works with floating point arguments:

```
>>> import numpy as n
>>> a = n.array([10, 20, 30, 40])
>>> a
array([10, 20, 30, 40])
>>> a = n.array([[1, 2, 3], [4, 5, 6]])
>>> a
n.array([[1, 2, 3],
          [4, 5, 6]])
>>> n.arange(1, 5, 0.5)
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
```

skipping ranges

Note that `range()`, `xrange()`, and `xrange()` can all skip values in a range if you give a third argument.

```
>>> range(1,20,3)
[1, 4, 7, 10, 13, 16, 19]
>>> [i for i in xrange(1,20,3)]
[1, 4, 7, 10, 13, 16, 19]
>>> n.arange(1,20,3)
array([ 1,  4,  7, 10, 13, 16, 19])
```

It must be an integer.

floating point ranges

Using `arange` for floating point can be risky, because you don't always know exactly how many values it will return (due to inaccuracies in floating point math). Instead you can use `linspace` which takes a start value, an end value, and how many values there should be as arguments. This is frequently used for evaluating functions over a set of points.

```
>>> linspace(0, 2, 9)
array([ 0. ,  0.25,  0.5 ,  0.75,  1. ,  1.25,  1.5 ,  1.75,  2. ])
>>> x = n.linspace(0,2*math.pi,9)
>>> y = n.sin(x)
>>> y
array([ 0.00000000e+00,  7.07106781e-01,  1.00000000e+00,
       7.07106781e-01,  1.22460635e-16, -7.07106781e-01,
      -1.00000000e+00, -7.07106781e-01, -2.44921271e-16])
```

Note that I'm using `numpy.sin`, not `math.sin` here. This operates elementwise. `math.sin` only works on scalars.

from list comprehensions.

```
>>> import numpy as n  
>>> a = n.array([n.range(i,i+4) for i in n.arange(1,5)])  
>>> a  
array([[1, 2, 3, 4],  
       [2, 3, 4, 5],  
       [3, 4, 5, 6],  
       [4, 5, 6, 7]])
```

you can also use range instead of n.arange in this case

When you iterate over an array, it happens by default with respect to the first axis (i.e. each iteration returns a row). If you want to iterate over one element at a time, use a.flat.

or generator expressions

Note that forming an array from a generator expression will not cause the generator to be evaluated; the result will itself be a generator.

```
>>> import numpy as n  
>>> a = n.array(n.arange(i,i+4) for i in n.arange(1,5))  
>>> a  
array(<generator object <genexpr> at 0xb6488e64>, dtype=object)
```

Generators are weird; we're going to just deal with list comprehensions for now.

Operations on arrays

Operations on two arrays are typically performed element-wise:

```
>>> a = n.array([1, 2, 3])
>>> b = n.array([4, 5, 6])
>>> a + b
array([5, 7, 9])
>>> a * b
array([4,10,18])
```

It's a good idea to make sure that your two arrays are the same size. If they're not, NumPy will yell at you. Also, remember that even operations like multiplication occur elementwise by default.

Special array-crafting functions

There are other functions to create arrays. Many of them take as arguments the dimension of the new array. This should be a tuple, and has to be wrapped in parentheses to distinguish it from multiple arguments.

```
>>> n.ones((2, 3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> n.ones(2,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/System/Library/Frameworks/Python.framework/Versions/2.5/Extras/lib
    a = empty(shape, dtype, order)
TypeError: data type not understood
```

The other functions to create arrays like this are `zeros`, which returns an array of zeros.

Special array-crafting functions

and `empty` which makes no attempt to initialize the contents of the array, so its values are random and based on the state of memory.

```
>>> n.empty((3,4))
array([[ -3.17556323e-042,  2.19432581e-314,  1.45669514e-266,
       -1.24294082e-043],
       [ 6.33886224e-321,  1.20953760e-312,  1.29760124e-266,
        1.00000000e+000],
       [ 7.00982212e-313, -3.17536431e-042,  1.00000000e+008,
        1.66003522e-266]])
```

```
>>> n.empty((3,4))
array([[ 1.71626670e-266,  7.23813523e-316,  0.00000000e+000,
       -1.24294079e-043],
       [ 6.34874355e-321,  1.20953760e-312, -3.17543186e-042,
        0.00000000e+000],
       [ 0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        1.00000000e+000]])
```

Don't rely on this for true randomness of course.

Array creation options

You can specify the type of array at time of creation

```
>>> n.zeros((3,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> n.zeros((3,4),dtype=int)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
>>> n.zeros((3,4),dtype=complex)
array([[ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]])
```

Array creation options

You can create an n-dimentional array from a function of n parameters

```
>>> a = np.fromfunction(lambda x,y:x*y,(5,4))  
>>> a  
array([[ 0.,  0.,  0.,  0.],  
       [ 0.,  1.,  2.,  3.],  
       [ 0.,  2.,  4.,  6.],  
       [ 0.,  3.,  6.,  9.],  
       [ 0.,  4.,  8., 12.]])
```

Printing Arrays

```
>>> print n.ones((5,8),dtype=int)
[[1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]]
```



```
>>> print n.ones((80,70),dtype=int)
[[1 1 1 ..., 1 1 1]
 [1 1 1 ..., 1 1 1]
 [1 1 1 ..., 1 1 1]
 ...
 [1 1 1 ..., 1 1 1]
 [1 1 1 ..., 1 1 1]
 [1 1 1 ..., 1 1 1]]
```

hey look, it prints it like a table and skips rows and columns when there are too many! But you can change this!

```
>>> n.set_printoptions(threshold=9000)
>>> print n.empty((80,70), dtype=int)
```

Array shape

Arrays have a shape, which is a tuple specifying the length in each dimension. You can change that either by changing the shape variable (which operates in place) or by using the reshape function.

```
>>> a = n.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a.shape = 3,4
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.reshape(12)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a.reshape(2, 2, 3)
array([[[ 0,  1,  2],
       [ 3,  4,  5],
       [[ 6,  7,  8],
       [ 9, 10, 11]]]])
```

To see how many dimensions an array has, use either `ndim` or `rank`. Use either the `size` function or the `size` attribute to see the total number of elements.

```
>>> a = arange(12).reshape(3, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> rank(a)
2
>>> a.ndim
2
>>> size(a)
12
>>> a.size
12
```

```
>>> a = n.arange(36)
>>> a.reshape(2,2,3,3)
array([[[[ 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]]]])
```

flattening

look: its flat! look: its transposed!

```
>>> a = n.fromfunction(lambda x,y:x*(y+1),(5,4))>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  2.,  3.,  4.],
       [ 2.,  4.,  6.,  8.],
       [ 3.,  6.,  9., 12.],
       [ 4.,  8., 12., 16.]])
>>> a.flatten()
array([ 0.,  0.,  0., ...,  8., 12., 16.])
>>> a.transpose()
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 0.,  2.,  4.,  6.,  8.],
       [ 0.,  3.,  6.,  9., 12.],
       [ 0.,  4.,  8., 12., 16.]])
```

turtle stacking

numpy.vstack takes a sequence of arrays and turns them into an array with one stacked on top of the other.

```
>>> a = n.array([1,2,3])
>>> b = n.array([5,6,7])
>>> n.vstack((a,b))
array([[1, 2, 3],
       [5, 6, 7]])
>>> n.vstack((a,n.vstack((a,b))))
array([[1, 2, 3],
       [1, 2, 3],
       [5, 6, 7]])
>>> n.vstack([n.arange(i,i+12,2) for i in xrange(3,9)])
array([[ 3,  5,  7,  9, 11, 13],
       [ 4,  6,  8, 10, 12, 14],
       [ 5,  7,  9, 11, 13, 15],
       [ 6,  8, 10, 12, 14, 16],
       [ 7,  9, 11, 13, 15, 17],
       [ 8, 10, 12, 14, 16, 18]])
```

shoulder-to-shoulder

numpy.hstack sticks them end to end.

```
>>> ([n.arange(i,i+12,2).reshape(2,3) for i in xrange(3,9)])
[array([[ 3,  5,  7],
       [ 9, 11, 13]]), array([[ 4,  6,  8],
       [10, 12, 14]]), array([[ 5,  7,  9],
       [11, 13, 15]]), array([[ 6,  8, 10],
       [12, 14, 16]]), array([[ 7,  9, 11],
       [13, 15, 17]]), array([[ 8, 10, 12],
       [14, 16, 18]]])
>>> n.hstack([n.arange(i,i+12,2).reshape(2,3) for i in xrange(3,9)])
array([[ 3,  5,  7, ...,  8, 10, 12],
       [ 9, 11, 13, ..., 14, 16, 18]])
```

Indexing

You can index over arrays, using commas to separate dimensions

```
>>> a = arange(12).reshape(3, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[1]
array([4, 5, 6, 7])
>>> a[1, 2]
6
>>> a[:,2]
array([ 2,  6, 10])
>>> a[2,:]
array([ 8,  9, 10, 11])
```

Indexing with boolean arrays

You can manually pick which array elements you want. Checkerboard pattern

```
>>> c = n.fromfunction(lambda x,y:(x+y)%2 ==0,(4,4))
>>> c
array([[ True, False, True, False],
       [False, True, False, True],
       [ True, False, True, False],
       [False, True, False, True]], dtype=bool)
>>> a = n.fromfunction(lambda x,y:x+y,(4,4))
>>> a
array([[ 0.,  1.,  2.,  3.],
       [ 1.,  2.,  3.,  4.],
       [ 2.,  3.,  4.,  5.],
       [ 3.,  4.,  5.,  6.]])
>>> a[c]
array([ 0.,  2.,  2.,  4.,  2.,  4.,  4.,  6.])
```

Matrix mathematics

If you want to do matrix math, use either the dot function or create matrix objects (which are similar to arrays, but act like matrices in some ways)

```
>>> A = n.array([[1, 1], [0, 1]])
>>> B = n.array([[2, 0], [3, 4]])
>>> A * B
array([[2, 0],
       [0, 4]])
>>> n.dot(A, B)
array([[5, 4],
       [3, 4]])
>>> n.mat(A) * n.mat(B)
matrix([[5, 4],
       [3, 4]])
```

SciPy in turn creates lots of useful matrix functions. As with NumPy, you probably want to run `import scipy as s` just for convenience.

Linear Algebra

If you create matrices just like arrays with `matrix()`, you can do matrix multiplication. There are also some useful functions in `scipy.linalg`, such as `norm()`, `inv()`, `solve()`, `det`, and `eig`.
Let's model the set of equations:

$$x + y + z = 1$$

$$4x + 4y + 3z = 2$$

$$7x + 8y + 5z = 1$$

```
>>> A = matrix([[1, 1, 1], [4, 4, 3], [7, 8, 5]])
>>> b = matrix([[1, 2, 1]]).transpose()
>>> print linalg.inv(A) * b
[[ 1.]
 [-2.]
 [ 2.]]
```

Polynomials

There is a whole range of functions for manipulating polynomials. To create a polynomial, use the `poly1d()` function, which takes as an argument a list of polynomial coefficients.

```
>>> p = poly1d([3, 4, 5])
>>> print p
2
3 x + 4 x + 5
>>> print p*p
4      3      2
9 x + 24 x + 46 x + 40 x + 25
>>> p(2)
25
>>> print p.deriv()

6 x + 4
>>> print p.integ()
3      2
1 x + 2 x + 5 x
```

Numerical Integration

Instead of integrating polynomials, if you want to integrate numerically, define a function and then use `scipy.integrate.quad`, which takes three arguments: the function, the beginning of the integral, and the end of the integral.

Other related Modules to look at:

Matplotlib: creating graphs from numpy array data. Scipy: Scientific functions including integration, fast fourier transforms, signal processing, statistics, image processing. Divisi: AI library for reasoning by analogy.

