Numpy

March 23, 2016

1 Numpy

- Python extensions for manipulating large sets of objects organised in a grid-like fashion
- Vectors
- Matrices
- etc...
- Normal Python datastructures too slow

2 Using numpy

```
In [2]: import numpy as np
```

3 Array objects

- Homogeneous collection of large numbers of numbers
- Numbers of the same type
- Array objects must be full
- Size immutable
- Numbers can change
- size = total number of elements in the array (Does not change)
- shape = number of dimensions

4 Array objects

- rank = len(shape)
- typecode = single character identifying the element kind
- Number format (i, d etc)
- Character
- Python reference
- itemsize = number 8-bit bytes representing a single element

5 Creating arrays from scratch

Creating arrays with explicit type codes:

```
In [4]: a = np.array([1, 2, 5], float)
        print(a)
```

```
[ 1.  2.  5.]
```

```
In [3]: b = np.array([5, 4, 3], int)
        print(b)
```

```
[5 4 3]
```

When creating arrays without type codes, Python will figure out and select type from the provided data.

```
In [4]: c = np.array([1.0, 1.5, 3.0])
        print(c)
```

```
[ 1.   1.5  3. ]
```

```
In [5]: d = np.array([1,2,3,4,5,6])
        print(d)
```

```
[1 2 3 4 5 6]
```

6 Multidimensional arrays

```
In [6]: a = np.array([[1,2],[3,4]])
        print(a)
```

```
[[1 2]
 [3 4]]
```

```
In [7]: b = np.array([[1,2,3,4],[5,6,7,8]], float)
        print(b)
```

```
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]]
```

The shape of the array can be queried using the shape attribute:

```
In [8]: print(a.shape)
```

```
(2, 2)
```

```
In [10]: print(b.shape)
         r, c = b.shape
         print(r, c)
```

```
(2, 4)
2 4
```

7 Reshaping arrays

```
In [11]: a = np.array([[1,2],[3,4]])
         print(a)
```

```
[[1 2]
 [3 4]]
```

```
In [12]: a_flat = np.reshape(a, [4,])
         print(a_flat)
```

```
[1 2 3 4]
```

```
In [13]: a_flat = np.reshape(a, [1,4])
         print(a_flat)
```

```
[[1 2 3 4]]
```

```
In [14]: a_flat = np.reshape(a, [4,1])
         print(a_flat)
```

```
[[1]
 [2]
 [3]
 [4]]
```

```
In [15]: b = np.array([[1,2,3,4],[5,6,7,8]], float)
         print(b)
```

```
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]]
```

```
In [16]: b_shaped = np.reshape(b, [8,])
         print(b_shaped)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.]
```

```
In [17]: b_shaped = np.reshape(b, [4,2])
         print(b_shaped)
```

```
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]
```

Please note this is not the same as:

```
In [18]: b_trans = np.transpose(b)
         print(b_trans)
```

```
[[ 1.  5.]
 [ 2.  6.]
 [ 3.  7.]
 [ 4.  8.]]
```

8 Growing an array

```
In [19]: a = np.array([1,2])
         print(a)
```

```
[1 2]
```

```
In [20]: b = np.array([a,a])
         print(b)
```

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

```
In [21]: base = np.array([[1,2],[3,4]])
         print(base)
```

```
[[1 2]
 [3 4]]
```

```
In [22]: big = np.resize(base, [9,9])
         print(big)
```

```
[[1 2 3 4 1 2 3 4 1]
 [2 3 4 1 2 3 4 1 2]
 [3 4 1 2 3 4 1 2 3]
 [4 1 2 3 4 1 2 3 4]
 [1 2 3 4 1 2 3 4 1]
 [2 3 4 1 2 3 4 1 2]
 [3 4 1 2 3 4 1 2 3]
 [4 1 2 3 4 1 2 3 4]
 [1 2 3 4 1 2 3 4 1]]
```

```
In [23]: big = np.resize(base, [4,4])
         print(big)
```

```
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
```

```
In [24]: big = np.resize(base, [4,2])
         print(big)
```

```
[[1 2]
 [3 4]
 [1 2]
 [3 4]]
```

9 Arrays on the fly

```
In [25]: a = np.zeros([4,4])
         print(a)
```

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

```
In [26]: b = np.ones([5,10], float)
         print(b)
```

```
[[ 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.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]]
```

```
In [27]: a = np.arange(10)
         print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
In [28]: b = np.reshape(np.arange(100), [10,10])
         print(b)
```

```
[[ 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]
 [90 91 92 93 94 95 96 97 98 99]]
```

```
In [29]: a = np.arange(0,10)
         print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
In [30]: a = np.arange(-10,10)
         print(a)
```

```
[-10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7
   8   9]
```

```
In [31]: a = np.arange(-10,10,2)
         print(a)
```

```
[-10  -8  -6  -4  -2   0   2   4   6   8]
```

There are different ways of creating arrays with specific numbers. However, it is important to do it in the correct way. First the slow way:

```
In [32]: a = np.array([[42]*5]*5)
         print(a)
```

```
[[42 42 42 42 42]
 [42 42 42 42 42]
 [42 42 42 42 42]
 [42 42 42 42 42]
 [42 42 42 42 42]]
```

First creating a zero array and then adding 42 is much faster:

```
In [35]: a = np.zeros([5,5])+42
         print(a)
```

```
[[ 42.  42.  42.  42.  42.]
 [ 42.  42.  42.  42.  42.]
 [ 42.  42.  42.  42.  42.]
 [ 42.  42.  42.  42.  42.]
 [ 42.  42.  42.  42.  42.]]
```

Identity arrays can also be created using the identity() function.

```
In [38]: i = np.identity(10)
         print(i)
```

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
```

10 Creating arrays with a linear variation

```
In [39]: x = np.linspace(0,1.0,10)
         print(x)
```

```
[ 0.          0.11111111  0.22222222  0.33333333  0.44444444  0.55555556
  0.66666667  0.77777778  0.88888889  1.        ]
```

```
In [40]: x = np.linspace(0,1.0,20)
         print(x)
```

```
[ 0.          0.05263158  0.10526316  0.15789474  0.21052632  0.26315789
  0.31578947  0.36842105  0.42105263  0.47368421  0.52631579  0.57894737
  0.63157895  0.68421053  0.73684211  0.78947368  0.84210526  0.89473684
  0.94736842  1.        ]
```

11 Operating on arrays

Most python operators can be used with arrays. The following examples shows how it can be used.

```
In [41]: a = np.array([[1,2],[3,4]])
         print(a)
```

```
[[1 2]
 [3 4]]
```

Element wise addition.

```
In [42]: print(a+3)
```

```
[[4 5]
 [6 7]]
```

Element wise multiplication

```
In [43]: print(a*3)
```

```
[[ 3  6]
 [ 9 12]]
```

It is also possible to use normal mathematical functions with arrays.

```
In [44]: print(np.sin(a))
```

```
[[ 0.84147098  0.90929743]
 [ 0.14112001 -0.7568025 ]]
```

Element wise negation using the - operator.

```
In [45]: print(-a)
```

```
[[-1 -2]
 [-3 -4]]
```

Arrays of the same number of elements can also be added together.

```
In [46]: print(a+a)
```

```
[[2 4]
 [6 8]]
```

Adding differently sized arrays is also possible. Values from the smaller array is continously added to the larger array as shown in this example:

```
In [47]: a = np.array([1,2,3])
         b = np.ones([5,3])
         print(a)
         print(b)
```

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

```
In [48]: print(a+b)
```

```
[[ 2.  3.  4.]
 [ 2.  3.  4.]
 [ 2.  3.  4.]
 [ 2.  3.  4.]
 [ 2.  3.  4.]]
```

12 Getting and setting values

Retrieving a single value:

```
In [49]: a = np.arange(10)
         print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
In [52]: print(a[0])
```

```
0
```

Retrieving a range of values:

```
In [53]: print(a[1:5])
```

```
[1 2 3 4]
```

Please note that the range index is defined as [start:stop] and does not include the *stop* index.
You can get all values of a list except the last one by using -1 as the last index.

```
In [58]: print(a[:-1])
```

```
[0 1 2 3 4 5 6 7 8]
```

Using other negative numbers will retrieve other ranges:

```
In [59]: print(a[:-2])
```

```
[0 1 2 3 4 5 6 7]
```

It is also possible to use arange() and reshape to create arrays with increasing numbers:

```
In [60]: a = np.arange(16)+1
         print(a)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
```

```
In [61]: b = np.reshape(a, [4,4])
         print(b)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

It is also possible to retrieve rows and columns:

```
In [62]: print(b[0]) # row 0 of b
```

```
[1 2 3 4]
```

```
In [63]: print(b[:,1]) # column 1 of b
```

```
[ 2  6 10 14]
```

```
In [64]: print(b[-1]) # last row of b
```

```
[13 14 15 16]
```

```
In [65]: print(b[:,-2]) # column 2 of b
```

```
[ 3  7 11 15]
```

```
In [66]: print(b)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

Values can be assigned using index notation as well:

```
In [67]: b[0,0] = 42
         print(b)
```

```
[[42  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

Lists and arrays can be intermixed and used in assignments. Assign a row using the following statement:

```
In [68]: b[1] = [42,42,42,42]
         print(b)
```

```
[[42  2  3  4]
 [42 42 42 42]
 [ 9 10 11 12]
 [13 14 15 16]]
```

Columns can be assigned in the same fashion:

```
In [69]: b[:,2] = [42,42,42,42]
         print(b)
```

```
[[42  2 42  4]
 [42 42 42 42]
 [ 9 10 42 12]
 [13 14 42 16]]
```

13 Slicing

```
In [70]: a = np.reshape(np.arange(16)+1,[4,4])
         print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

```
In [71]: print(a[:,:])
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

```
In [72]: print(a[:,1])
```

```
[ 2  6 10 14]
```

```
In [73]: print(a[1,:])
```

```
[5 6 7 8]
```

14 Ufuncs

- Operates elementwise on arrays
- Available as callable objects (functions)
- Can operate on Python sequences
- Can take output arguments
- Have special methods

```
In [74]: a = np.arange(10)
         print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
In [ ]: print(np.add(a,a))
```

```
In [ ]: print(a+a)
```

```
In [ ]: print(np.sin(a))
```

```
In [ ]: print(add(a, range(10)))
```

Ufuncs are often faster, but source code will be harder to read. Sometimes using a ufunc can eliminate a unecessary copying of large arrays. The following example shows how a ufunc is used to avoid this.

```
In [75]: a = np.arange(10)
         a = a * 10 # This will make a copy of a
         print(a)
```

```
[ 0 10 20 30 40 50 60 70 80 90]
```

Again with a ufunc.

```
In [76]: a = np.arange(10)
         np.multiply(a,10,a)
         print(a)
```

```
[ 0 10 20 30 40 50 60 70 80 90]
```

14.1 The add.reduce function (A quicker sum)

add.reduce() can be used to sum (reduce) an array.

```
In [ ]: a = np.arange(10)
        print(a)
```

```
In [ ]: print(np.add.reduce(a))
```

This is faster than:

```
In [ ]: print(a.sum())
```

14.2 add.accumulate

```
In [ ]: a = np.arange(10)
        print(a)
```

```
In [ ]: print(np.add.accumulate(a))
```

14.3 transpose

To do a matrix-transpos of an array, the transpose() function can be used.

```
In [ ]: a = np.reshape(np.arange(16), [4,4])
        print(a)
```

```
In [ ]: print(np.transpose(a))
```

14.4 diagonal() and trace() functions

```
In [77]: a = np.reshape(np.arange(16), [4,4])
         print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
In [78]: print(np.diagonal(a))
```

```
[ 0  5 10 15]
```

```
In [79]: print(np.trace(a))
```

```
30
```

It is also possible to do traces left and right of the diagonal, by adding a parameter to the trace() function.

```
In [80]: print(np.trace(a,1))
```

```
18
```

```
In [81]: print(np.trace(a,-1))
```

```
27
```

14.5 Matrix multiplication

The * operator on array is equivalent to an elementwise multiplication of matrices. If you want to do a matrix multiplication on arrays, .dot() method or the @-operator can be used.

```
In [82]: a = np.reshape(np.arange(16), [4,4])
         print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
In [83]: print(a.dot(a))
```

```
[[ 56  62  68  74]
 [152 174 196 218]
 [248 286 324 362]
 [344 398 452 506]]
```

15 Exempel 1

```
In [84]: import math
         import numpy as np

         x = np.linspace(0.0, 2.0*math.pi, 20)
         y = np.sin(x)

         print(x)
         print(y)
```

```
[ 0.          0.33069396  0.66138793  0.99208189  1.32277585  1.65346982
  1.98416378  2.31485774  2.64555171  2.97624567  3.30693964  3.6376336
  3.96832756  4.29902153  4.62971549  4.96040945  5.29110342  5.62179738
  5.95249134  6.28318531]
[  0.00000000e+00   3.24699469e-01   6.14212713e-01   8.37166478e-01
   9.69400266e-01   9.96584493e-01   9.15773327e-01   7.35723911e-01
   4.75947393e-01   1.64594590e-01  -1.64594590e-01  -4.75947393e-01
  -7.35723911e-01  -9.15773327e-01  -9.96584493e-01  -9.69400266e-01
  -8.37166478e-01  -6.14212713e-01  -3.24699469e-01  -2.44929360e-16]
```

16 Exempel 2

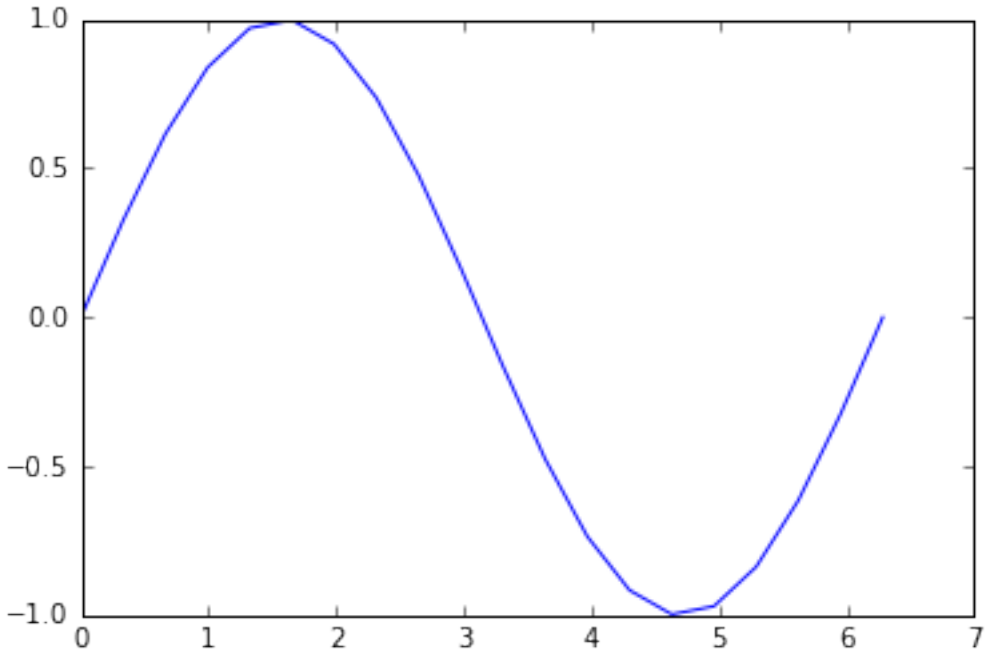
```
In [85]: import pylab as pl

In [86]: %matplotlib inline

In [88]: x = np.linspace(0.0, 2.0*math.pi, 20)
         y = np.sin(x)

         pl.plot(x,y)

Out[88]: [<matplotlib.lines.Line2D at 0x1089952b0>]
```



17 Matrices in Numpy

Arrays in Numpy are a generic array storage type and in itself not aware of any matrix operations. If this is desired Numpy provides an array derived type matrix, which provides this functionality. Some examples are illustrated below:

```
In [89]: A = np.matrix( [[1,2,3],[11,12,13],[21,22,23]])
         x = np.matrix( [[1],[2],[3]] )
         y = np.matrix( [[1,2,3]] )
         print(A)

[[ 1  2  3]
 [11 12 13]
 [21 22 23]]

In [90]: print(x)

[[1]
 [2]
 [3]]

In [91]: print(y)

[[1 2 3]]

In [92]: print(A.T) # Matrix transpose

[[ 1 11 21]
 [ 2 12 22]
 [ 3 13 23]]

In [93]: print(A*x) # Matrix mulitply A * x

[[ 14]
 [ 74]
 [134]]

In [94]: print(A.I) # Matrix inverse

[[  2.60538821e+14  -5.21077643e+14   2.60538821e+14]
 [ -5.21077643e+14   1.04215529e+15  -5.21077643e+14]
 [  2.60538821e+14  -5.21077643e+14   2.60538821e+14]]

In [95]: print(np.linalg.solve(A,x)) # Solve A * y = x

[[-0.01900826]
 [-0.66198347]
 [ 0.78099174]]

In [ ]:
```