NumPy Primer

David Warde-Farley

February 4, 2014

0.1 What is NumPy?

Python is a fairly good language for scientific work. It is easy to learn, and can easily interact with tons of different things. But, insofar as most scientific computing is numerical in nature, it has a few problems:

- It is slow. In particular, function call overhead is quite high, and loops are quite slow.
- Its basic data structures are optimized for use cases other than storing large quantities of numerical data. The standard library `array` module is an option but it is clunky and not very featureful.
- Interacting with big huge nested lists and dictionaries can get quite complicated rather quickly.

NumPy is a library that provides a powerful N-dimensional array container that gets around some of these problems.

Diving In

At first glance, NumPy arrays appear similar to (nested) Python sequences:

```
In [1]: import numpy as np
        np.array([0, 2, 4])

Out[1]: array([0, 2, 4])

In [2]: np.array([[2, 9, 3], [2, 5, 7]])

Out[2]: array([[2, 9, 3],
               [2, 5, 7]])
```

Things are a little bit subtler, however.

```
In [3]: a = np.array([[5, 2], [3, 9]])

In [4]: a.dtype

Out[4]: dtype('int64')

In [5]: b = np.array([[5., 2], [3, 9]])

In [6]: b

Out[6]: array([[ 5.,  2.],
               [ 3.,  9.]])

In [7]: b.dtype

Out[7]: dtype('float64')
```

NumPy arrays in a nutshell

NumPy arrays are

- *explicitly typed** (every element shares a common, usually numeric, machine-level type),
- homogeneous* (only one type of atomic element per array),
- N-dimensional grids (no ragged dimensions – necessary for rapid index computations)
- Backed by a real chunk of memory

Fundamentally important attributes:

- `ndim` – the number of dimensions
- `shape` – a tuple of integers; the length along each dimension (`len(x.shape) == x.ndim`)
- `dtype` – a descriptor of the individual elements of the array

dtypes in more detail
Unlike Python builtin types, where integers can get as large as you like (but get slower past a certain point), types NumPy behave more like they do in languages that are "closer to the metal".
Built-in NumPy numeric dtypes:

- `int8`, `int16`, `int32`, `int64`, `uint*` variants
- `float32`, `float64` (a.k.a. single and double precision, or `float` and `double` in other languages)
- `bool`
- string/Unicode (with bounded character length) and object arrays are also supported

In all multi-byte cases, NumPy supports both big- and little-endian modes (if you don't know what this is, don't worry about it).
Compound dtypes can be created, with field names:

```
In [8]: dt = np.dtype([('name', (str, 8)), ('age', 'int8')])
        np.array([('David Warde-Farley', 27), ('Joan', 70)], dtype=dt)

Out[8]: array([('David Wa', 27), ('Joan', 70)],
              dtype=[('name', 'S8'), ('age', 'i1')])

In [9]: arr = _; arr['age']

Out[9]: array([27, 70], dtype=int8)
```

Can be quite powerful for tabular data; integrates with PyTables HDF5 tables. Pandas pushes this idea even further with a (NumPy-based) object called the DataFrame.
In these notes we will focus on numeric arrays only.

0.2 Other ways to create NumPy arrays

An array full of zeros, of a given dtype:

```
In [10]: np.zeros((2, 3), dtype=np.float32) # all of these functions take a dtype keyword

Out[10]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]], dtype=float32)
```

An empty (uninitialized!) array of 16-bit integers:

```
In [11]: np.empty((2, 5), dtype='>i2')  # terse string codes too

Out[11]: array([[-22393, -27187,  24703,      0, -22393],
                [-27187,  24703,      0,   8192,      0]], dtype=int16)
```

0.3 Even more ways to create NumPy arrays

The identity matrix (complex dtype):

```
In [12]: np.eye(4, dtype='complex64') # Identity matrix
```

```
Out[12]: array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
                [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
                [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
                [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]], dtype=complex64)
```

```
In [13]: type((25,))
```

```
Out[13]: tuple
```

```
In [14]: np.arange(3, 10, 2) # Same arguments, behaviour as range()/xrange()
```

```
Out[14]: array([3, 5, 7, 9])
```

```
In [15]: np.linspace(0, 1, 5) # 5 values linearly between 0 and 1 inclusive
```

```
Out[15]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
In [16]: np.logspace(-5, 0, 3) # args are exponent (base 10, or `base` keyword)
```

```
Out[16]: array([  1.00000000e-05,   3.16227766e-03,   1.00000000e+00])
```

ufuncs
NumPy includes many *universal functions*, or *ufuncs*, that operate on every element in an array. The loop is performed in C, so it's quite fast.

```
In [17]: np.sqrt(np.arange(20))
```

```
Out[17]: array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
                 2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ,
                 3.16227766,  3.31662479,  3.46410162,  3.60555128,  3.74165739,
                 3.87298335,  4.        ,  4.12310563,  4.24264069,  4.35889894])
```

Basic indexing/slicing
Another big difference between NumPy arrays and nested sequences: much more powerful indexing.

```
In [18]: a = np.random.uniform(size=(5, 3, 2))
         a[0, 0, 0]
```

```
Out[18]: 0.85655565874766459
```

```
In [19]: a[0, 0, :]
```

```
Out[19]: array([ 0.85655566,  0.51002294])
```

```
In [20]: a[0, :2, :]
```

```
Out[20]: array([[ 0.85655566,  0.51002294],
                [ 0.7466188 ,  0.72895747]])
```

```
In [21]: a[1:5:2, 2, :]  # Mixing single index, slicing
```

```
Out[21]: array([[ 0.07556729,  0.45407778],
                [ 0.74635885,  0.88126337]])
```

Views

Note that when slicing and indexing arrays like with single indices and slices, *no data copy is performed*. For example:

```
In [22]: b = a[0, ::2, :]
         print b.shape
```

```
(2, 2)
```

```
In [23]: b[0, 0] = 5; b[0, 1] = 6; b[1, 0] = 7; b[1, 1] = 8
```

Now, if we look at the same slice of a:

```
In [24]: a
```

```
Out[24]: array([[[ 5.        ,  6.        ],
                 [ 0.7466188 ,  0.72895747],
                 [ 7.        ,  8.        ]],

                [[ 0.92056328,  0.2326089 ],
                 [ 0.84954143,  0.94593019],
                 [ 0.07556729,  0.45407778]],

                [[ 0.28246929,  0.43871352],
                 [ 0.171978  ,  0.96171135],
                 [ 0.99691723,  0.07196666]],

                [[ 0.90508994,  0.58769137],
                 [ 0.38192355,  0.42424631],
                 [ 0.74635885,  0.88126337]],

                [[ 0.81731908,  0.7014133 ],
                 [ 0.52548962,  0.65010616],
                 [ 0.77429617,  0.77658705]]])
```

This is a very good thing for memory efficiency and speed, but can lead to unexpected bugs for newcomers. The copy method is your friend.

0.4 Exercise 1

```
In [25]: a = np.array([[ 1,  2,  3,  4],
                       [ 5,  6,  7,  8],
                       [ 9, 10, 11, 12],
                       [13, 14, 15, 16]])
         # Or...
         a = np.arange(1, 17).reshape((4, 4))
```

Index into the array to pull out the 2x2 sub-array corresponding to 2, 3, 6 and 7.

```
In [26]: # SOLUTION TO EXERCISE 1

         a[:2, 1:3]
```

```
Out[26]: array([[2, 3],
                [6, 7]])
```

Now, set the 9, 10, 11, 13, 14 and 15 elements to 0. Note that you can use these indexing expressions on the left hand side of an assignment.

```
In [27]: # SOLUTION TO EXERCISE 1b
         a[2:, :3] = 0
         # Display a
         a
```

```
Out[27]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 0,  0,  0, 12],
                [ 0,  0,  0, 16]])
```

Advanced Indexing

Advanced or “fancy” indexing allows one to use arrays or lists to indexing along a given dimension. In contrast to basic indexing and slicing, advanced indexing **always** creates a copy.

```
In [28]: b = np.arange(15).reshape((5, 3))
         b
```

```
Out[28]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])
```

```
In [29]: b[[0, 1, 4]] # Implicitly b[[0, 1, 4], :]
```

```
Out[29]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [12, 13, 14]])
```

Multiple axes and advanced indexing

```
In [30]: b[[1, 4], [0, 2]]
         b
```

```
Out[30]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])
```

We’re not limited to one-dimensional index arrays along each axis.

```
In [31]: b[[[1, 4], [2, 1]], [[2, 1], [1, 2]]]
```

```
Out[31]: array([[ 5, 13],
                [ 7,  5]])
```

Note that deterministic but unintuitive things happen when you mix advanced indexing and slicing. My advice: *just say no.*

Exercise 2

Pull out the values 1, 2, 3, 4, 5 from `c` in ascending order using advanced indexing.

```
In [32]: c = np.array([[4, 9, 2, 5],
                       [1, 6, 3, 7]])
```

```
In [33]: # SOLUTION TO EXERCISE 2

         c[[1, 0, 1, 0, 0], [0, 2, 2, 0, 3]]

Out[33]: array([1, 2, 3, 4, 5])
```

Elementwise operations

Part of the real power of NumPy lies in how easily and quickly (faster than Python loops) you can manipulate the values in whole arrays at once.

```
In [34]: 5 * np.arange(4) + 3

Out[34]: array([ 3,  8, 13, 18])
```

Arithmetic with scalars applies to the entire array. Array-array operations work as expected (elementwise) when things have the same shape.

```
In [35]: np.arange(4) | np.array([3, 5, 3, 1])

Out[35]: array([3, 5, 3, 3])
```

Broadcasting

When two arrays **don't** have the same shape, NumPy will attempt to **broadcast** them to the same shape. This first and foremost means that any dimensions with length 1 in one array are **repeated along that axis** for the purposes of the operation taking place. This is best illustrated with an example.

```
In [36]: row = np.array([[3, 6, 9]])
         row.shape

Out[36]: (1, 3)

In [37]: identity = np.eye(3)
         identity

Out[37]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In [38]: row.shape

Out[38]: (1, 3)

In [39]: identity.shape

Out[39]: (3, 3)

In [40]: row + identity

Out[40]: array([[  4.,   6.,   9.],
                [  3.,   7.,   9.],
                [  3.,   6.,  10.]])

In [41]: col = np.array([[3],
                         [6],
                         [9]])
         col.shape

Out[41]: (3, 1)
```

```
In [42]: col + identity

Out[42]: array([[  4.,   3.,   3.],
                [  6.,   7.,   6.],
                [  9.,   9.,  10.]])
```

More Broadcasting Fun

If two arrays in an operation do not have the same number of dimensions, NumPy follows a simple rule to try and broadcast them:

Left pad the shape of the array with less dimensions with 1s,then proceed as above.

(*) You can think of a scalar as a 0-dimensional array with shape `()`, and this unifies broadcasting and array-scalar operations.

```
In [43]: one_d = np.array([3, 6, 9])
         one_d.shape

Out[43]: (3,)

In [44]: one_d + np.eye(3)

Out[44]: array([[  4.,   6.,   9.],
                [  3.,   7.,   9.],
                [  3.,   6.,  10.]])
```

What if I want the other way?

Often you want to add singleton axes to an existing array of lower dimension. The easiest method is to index with `np.newaxis` (which is actually defined as `None`):

```
In [45]: one_d[:, np.newaxis].shape

Out[45]: (3, 1)

In [46]: one_d[:, np.newaxis] + np.eye(3)

Out[46]: array([[  4.,   3.,   3.],
                [  6.,   7.,   6.],
                [  9.,   9.,  10.]])
```

The `reshape` method/function also works.

Reductions

In many, many settings, we care about some property of a group of elements. We call these generally *reductions*.

NumPy arrays support a number of commonly required reductions as methods:

- `min`, `max` – Minimum and maximum element
- `argmin`, `argmax` – *index* of the minimum/maximum element
- `sum` – sum of elements
- `prod` – product of elements
- `mean` – mean element
- `var`, `std` – variance, standard deviation
- `all` – do all elements evaluate boolean `True`?
- `any` – do any elements evaluate boolean `True`?

They are also available as top-level functions in the `numpy` module.

Defaults to *all elements*, but an `axis` argument allows reduction over a given axis.

```
In [47]: r = np.random.uniform(size=(50, 20))
In [48]: r.mean()

Out[48]: 0.49783139983186347

In [49]: r.mean(axis=1)

Out[49]: array([ 0.45233148,  0.47911453,  0.56823188,  0.51976532,  0.46627496,
                 0.47032084,  0.46966941,  0.49628592,  0.51143339,  0.44615496,
                 0.5619306 ,  0.50798723,  0.5200812 ,  0.5117501 ,  0.5905414 ,
                 0.60000562,  0.48564358,  0.54050937,  0.43011936,  0.49141867,
                 0.46854121,  0.46278353,  0.50455998,  0.62494029,  0.60260209,
                 0.43352872,  0.55147621,  0.49192916,  0.46088348,  0.36207479,
                 0.50193017,  0.45710243,  0.53886275,  0.53566549,  0.41415739,
                 0.54524552,  0.51757536,  0.51686412,  0.40869995,  0.51117913,
                 0.52637351,  0.51756774,  0.43087067,  0.57645757,  0.46587674,
                 0.41310198,  0.44114834,  0.5681407 ,  0.4834228 ,  0.43843833])
```

Exercise 3

Write a function `normalize_vectors` that takes a 2D array and divides each row by the norm of that row, where the norm is defined as $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$.

```
In [50]: # SOLUTION TO EXERCISE 3

         def normalize_vectors(x):
             norms = np.sqrt((x ** 2).sum(axis=-1))
             return x / norms[..., np.newaxis]
```

Elementwise comparisons

Comparing NumPy arrays with each other, or with a scalar, operates elementwise on the arrays. All of the same broadcasting rules apply.

```
In [51]: a = np.array([[4, 9],
                       [5, 3]])

         a > 4

Out[51]: array([[False,  True],
                [ True, False]], dtype=bool)
```

Boolean arrays can be used for indexing, as well:

```
In [52]: a[a > 4]

Out[52]: array([9, 5])

In [53]: (a > 0).all()

Out[53]: True

In [54]: (a < 2).any()

Out[54]: False

In [55]: a

Out[55]: array([[4, 9],
                [5, 3]])

In [56]: a.mean(axis=0)

Out[56]: array([ 4.5,  6. ])
```