Using `Numpy` – Numerical Arrays in Python

Numpy provides a unified way to manage numerical arrays in Python. It grew out of many different approaches to handling numerical arrays. It is now the accepted way to compute with numerical arrays in Python and is the cornerstone of many other scientific computing Python modules. To understand and to be effective with scientific computing and Python, a solid grasp on Numpy is essential.

```
>>> import numpy as np # naming convention
```

dtypes

Although Python has a dynamically-typed system, Numpy provides the mechanics to specify types using `dtypes`

```
>>> a = np.array([0],np.int16) # 16-bit integer
>>> a.itemsize # in 8-bit bytes
2
>>> a.nbytes
2
>>> a = np.array([0],np.int64) # 64-bit integer
>>> a.itemsize
8
```

Arrays follow the same pattern,

```
>>> a = np.array([0,1,23,4],np.int64) # 64-bit integer
>>> a.shape
(4L,)
>>> a.nbytes
32
```

Note that you cannot tack on extra elements to a Numpy array after creation,

```
>>> a = np.array([1,2])
>>> a[2]=32
IndexError: index 2 is out of bounds for axis 0 with size 2
```

Also, once you create the array with a specific `dtype`, assigning to that array with cast to that type. For example,

```
>>> x=np.array(range(5),dtype=int)
>>> x[0] = 1.33 # float assignment does not match dtype=int
>>> x
array([1, 1, 2, 3, 4])
>>> x[0]='this is a string'
ValueError: invalid literal for long() with base 10: 'this is a string'
```

This if different from Matlab because the copy/view array semantics are so fundamentally different.

`datetime` dtypes

In addition to numeric and string `dtypes`, Numpy provides a `datetime` dtype. You can use a standard ISO datetime string representation as input,

```
>>> np.datetime64('2010-01-05')
numpy.datetime64('2010-01-05')
```

You can also force the resolution of the object as in the following,

```
>>> np.datetime64('2010-01-05','M')
numpy.datetime64('2010-01')
```

Note that the day was removed because `M` month was used as the resolution. Time can also be included as in the following,

```
>>> np.datetime64('2010-01-05T03:30')
>>> np.datetime64('2010-01-05T03:30:08.101')
```

Note the format of the `HH:MM:SS.ms` for hour, minutes, seconds, and fractions of a second.

It is more common to specify Numpy arrays,

```
>>> x = np.array(['2010-01-05','2010-01-06','2010-03-05T03:30'],dtype=np.datetime64)
>>> x
array(['2010-01-05T00:00', '2010-01-06T00:00', '2010-03-05T03:30'], dtype='datetime64[m]')
>>> x = np.array(['2010-01-05','2010-01-06','2010-03-05T03:30'],dtype='datetime64[M]')
>>> x
array(['2010-01', '2010-01', '2010-03'], dtype='datetime64[M]')
```

Notice that the last line specified the `dtype` as a string with the embedded month ('M') resolution. You can build a sequence of datetime64 objects using `np.arange`,

```
>>> np.arange('2005-01-01','2006-01-01',dtype='datetime64[M]')
array(['2005-01', '2005-02', '2005-03', '2005-04', '2005-05', '2005-06',
'2005-07', '2005-08', '2005-09', '2005-10', '2005-11', '2005-12'], dtype='datetime64[M]')
```

Again, notice how the resolution is specified in the string `dtype` specification.

Basic arithmetic operations are supported,

```
>>> np.datetime64('2010-01-01') - np.datetime64('2008-01-01')
numpy.timedelta64(731,'D')
```

Note that a `timedelta64` object was returned. Note that the resolution returned depends on the resolution of the arguments,

```
>>> np.datetime64('2010-01-01T03:30') - np.datetime64('2008-01-01T15:10')
numpy.timedelta64(1051940,'m')
```

You can create a `timedelta64` object directly and use it for

```
>>> np.timedelta64(10,'D') + np.datetime64('2010-10-10')
numpy.datetime64('2010-10-20')
```

Again, you can specify the resolution,

```
>>> np.timedelta64(10,'s') + np.datetime64('2010-10-10')
numpy.datetime64('2010-10-10T00:00:10')
```

Multiplication and division are supported where they make sense,

```
>>> np.timedelta64(80,'D')*3 + np.datetime64('2010-10-10')
numpy.datetime64('2011-06-07')
>>> np.timedelta64(1,'W') / np.timedelta64(1,'s')
604800.0
```

Now we know how many seconds are in a week!

Python already has a powerful `datetime` module. The easiest way to convert back and forth from Numpy datetime objects is with Pandas

```
>>> import pandas as pd
>>> k = np.datetime64('2010-10-10')
>>> pd.to_datetime(str(k))
Timestamp('2010-10-10 00:00:00')
```

Numpy datetime objects can also reason about weekends versus weekdays and some other calendar logic. The datetime64 documentation has more details.

Multidimensional arrays

Multidimensional arrays follow the same pattern

```
>>> a = np.array([[1,3],[4,5]]) # omitting dtype picks default
>>> a
array([[1, 3],
       [4, 5]])
>>> a.dtype
dtype('int32')
>>> a.shape
(2L, 2L)
>>> a.nbytes
16
>>> a.flatten()
array([1, 3, 4, 5])
```

You can have up to 32 dimensions. You need to compile Numpy yourself, if you want to increase this. Numpy offers many ways to build arrays automatically,

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = np.ones((2,2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> a = np.linspace(0,1,5)
>>> a
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
>>> X,Y = np.meshgrid([1,2,3],[5,6])
>>> X
array([[1, 2, 3],
       [1, 2, 3]])
>>> Y
array([[5, 5, 5],
       [6, 6, 6]])
>>> a = np.zeros((2,2))
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
```

You can create Numpy arrays using functions also,

```
>>> np.fromfunction(lambda i,j:abs(i-j)<=1,(4,4))
array([[ True,  True, False, False],
       [ True,  True,  True, False],
       [False,  True,  True,  True],
       [False, False,  True,  True]], dtype=bool)
```

Numpy arrays can have fieldnames also, but this is no longer common in codes in the wild.

```
>>> a = np.zeros((2,2),dtype=[('x','f4')])
>>> a['x']
array([[ 0.,  0.],
       [ 0.,  0.]], dtype=float32)
>>> x=np.array([(1,2)],dtype=[('value','f4'),('amount','c8')])
>>> x['value']
array([ 1.], dtype=float32)
>>> x['amount']
array([ 2.+0.j], dtype=complex64)
>>> x=np.array([(1,9),(2,10),(3,11),(4,14)],dtype=[('value','f4'),('amount','c8')])
>>> print x['value']
[ 1.  2.  3.  4.]
>>> print x['amount']
[  9.+0.j  10.+0.j  11.+0.j  14.+0.j]
```

Numpy arrays can also be accessed by their attributes using `recarray`,

```
>>> y = x.view(np.recarray)
>>> y.amount # access as attribute
array([  9.+0.j,  10.+0.j,  11.+0.j,  14.+0.j], dtype=complex64)
>>> y.value # access as attribute
array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Numpy Exercises

1. Create the following 5x5 upper triangular Numpy array.

```
array([[1, 1, 1, 1, 1],
       [0, 1, 1, 1, 1],
       [0, 0, 1, 1, 1],
       [0, 0, 0, 1, 1],
       [0, 0, 0, 0, 1]])
```

2. Create the following 5x5 upper triangular Numpy array.

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

Reshaping and stacking Numpy arrays

Arrays can be stacked horizontally and vertically

```
>>> x = np.arange(5)
>>> y = np.array([9,10,11,12,13])
>>> np.hstack([x,y])
array([ 0,  1,  2,  3,  4,  9, 10, 11, 12, 13])
>>> np.vstack([x,y])
array([[ 0,  1,  2,  3,  4],
       [ 9, 10, 11, 12, 13]])
```

There is also a `dstack` method if you want to stack in the third *depth* dimension. Numpy `np.concatenate` handles the general arbitrary-dimension case. In some codes (e.g., `scikit-learn`), you may find the terse `np.c_` and `np.r_` used to stack arrays column-wise and row-wise.

```
>>> np.c_[x,x] # column-wise
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])
>>> np.r_[x,x] # row-wise
array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

Duplicating Numpy Arrays

Numpy has a `repeat` function for duplicating elements and a more generalized version in `tile`,

```
>>> x=np.arange(4)
>>> np.repeat(x,2)
array([0, 0, 1, 1, 2, 2, 3, 3])
>>> np.tile(x,(2,1))
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
>>> np.tile(x,(2,2))
array([[0, 1, 2, 3, 0, 1, 2, 3],
       [0, 1, 2, 3, 0, 1, 2, 3]])
```

You can also have non-numerics like string as items in the array

```
>>> np.array(['a','b','cow','deep'])
array(['a', 'b', 'cow', 'deep'], dtype='|S4')
```

Note that the `'S4'` refers to string of length 4, which is the longest string in the sequence. Numpy has a `chararray` object for specialized character tools.

Reshaping Numpy arrays

Numpy arrays can be reshaped after creation.

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

For the truly lazy, you can replace one of the dimensions above by negative one (i.e., `reshape(-1,5)`) and Numpy will figure out the conforming other dimension. The transpose operation is the `.T` attribute,

```
>>> a.T
array([[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]])
```

The conjugate transpose (i.e., Hermitian transpose) is the `.H` attribute.

Slicing, logical array operations

Numpy arrays follow the same zero-indexed slicing logic as Python lists and strings.

```
>>> x = np.arange(50).reshape(5,10)
>>> x
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, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

The colon character means take all of the indicated dimension

```
>>> print x[:,0]
[ 0 10 20 30 40]
>>> print x[0,:]
[0 1 2 3 4 5 6 7 8 9]
>>> x = np.arange(50).reshape(5,10) # reshaping arrays
```

```
>>> print x
[[ 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]]

>>> print x[:,0] # any row, 0th column
[ 0 10 20 30 40]
>>> print x[0,:] # any column, 0th row
[0 1 2 3 4 5 6 7 8 9]
>>> x[1:3,4:6]
array([[14, 15],
       [24, 25]])
>>> x = np.arange(2*3*4).reshape(2,3,4) # reshaping arrays
>>> print x
[[[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]]
...
[[12 13 14 15]
[16 17 18 19]
[20 21 22 23]]]
>>> x[:,1,[2,1]] # index each dimension
array([[ 6,  5],
       [18, 17]])
```

Numpy's `where` function can find array elements according to specific logical criteria,

```
>>> np.where(x % 2 == 0)
(array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]), array([0, 0, 1, 1, 2, 2, 0, 0, 1, 1, 2, 2]), a
>>> x[ np.where(x % 2 == 0) ]
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22])
>>> x[ np.where( np.logical_and(x % 2 == 0,x < 9) ) ] # also logical_or, etc.
array([0, 2, 4, 6, 8])
```

Additionally, Numpy arrays can be indexed by logical Numpy arrays,

```
>>> a = np.arange(9).reshape((3,3))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b = np.fromfunction(lambda i,j:abs(i-j)<=1,(3,3))
array([[ True,  True, False],
       [ True,  True,  True],
       [False,  True,  True]], dtype=bool)
>>> a[b]
array([0, 1, 3, 4, 5, 7, 8])
>>> b = (a>4)
array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]], dtype=bool)
>>> a[b]
array([5, 6, 7, 8])
```

Numpy Arrays and Memory

Numpy uses pass-by-reference semantics so that slice operations are *views* into the array without implicit copying, which is consistent with Python's semantics. This is particularly helpful with large arrays that already strain available memory. In Numpy terminology, *slicing* creates views (no copying) and advanced indexing creates copies. Let's start with advanced indexing.

If the indexing object (i.e., the item between the brackets) is a non-tuple sequence object, another Numpy array (of type integer or boolean), or a tuple with at least one sequence object or Numpy array,

then indexing creates copies. For the above example, to accomplish the same array extension in Numpy, you have to do something like the following

```
>>> x = np.ones((3,3))
>>> x
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> x[:,[0,1,2,2]] # notice duplicated last dimension
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> y=x[:,[0,1,2,2]] # same as above, but do assign it to y
```

Because of advanced indexing, the variable `y` has its own memory because the relevant parts of `x` were copied. To prove it, we assign a new element to `x` and see that `y` is not updated.

```
>>> x[0,0]=999 # change element in x
>>> x                          # changed
array([[ 999.,    1.,    1.],
       [   1.,    1.,    1.],
       [   1.,    1.,    1.]])
>>> y                          # not changed!
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

However, if we start over and construct `y` by slicing (which makes it a view) as shown below, then the change we made *does* affect `y` because a view is just a window into the same memory.

```
>>> x = np.ones((3,3))
>>> y = x[:2,:2] # view of upper left piece
>>> x[0,0] = 999 # change value
>>> x
array([[ 999.,    1.,    1.], # see the change?
       [   1.,    1.,    1.],
       [   1.,    1.,    1.]])
>>> y
array([[ 999.,    1.], # changed y also!
       [   1.,    1.]])
```

Note that if you want to explicitly force a copy without any indexing tricks, you can do `y=x.copy()`. The code below works through another example of advanced indexing versus slicing.

```
>>> x = np.arange(5) # create array
>>> x
array([0, 1, 2, 3, 4])
>>> y=x[[0,1,2]] # index by integer list to force copy
>>> y
array([0, 1, 2])
>>> z=x[:3]      # slice creates view
>>> z            # note y and z have same entries
array([0, 1, 2])
>>> x[0]=999     # change element of x
>>> x
array([999,   1,   2,   3,   4])
>>> y            # note y is unaffected,
array([0, 1, 2])
>>> z            # but z is (it's a view).
array([999,   1,   2])
```

In this example, `y` is a copy, not a view, because it was created using advanced indexing whereas `z` was created using slicing. Thus, even though `y` and `z` have the same entries, only `z` is affected by changes to

`x`. Note that the `flags.ownsdata` property of Numpy arrays can help sort this out until you get used to it.

Viewing Memory of Numpy Arrays

Manipulating memory using views is particularly powerful for signal and image processing algorithms that require overlapping fragments of memory. The following is an example of how to use advanced Numpy to create overlapping blocks that do not actually consume additional memory,

```
>>> from numpy.lib.stride_tricks import as_strided
>>> x = arange(16)
>>> y=as_strided(x,(7,4),(8,4)) # overlapped entries
>>> y
array([[ 0,  1,  2,  3],
       [ 2,  3,  4,  5],
       [ 4,  5,  6,  7],
       [ 6,  7,  8,  9],
       [ 8,  9, 10, 11],
       [10, 11, 12, 13],
       [12, 13, 14, 15]])
```

The above code creates a range of integers and then overlaps the entries to create a `7x4` Numpy array. The final argument in the `as_strided` function are the strides, which are the steps in bytes to move in the row and column dimensions, respectively. Thus, the resulting array steps four bytes in the column dimension and eight bytes in the row dimension. Because the integer elements in the Numpy array are four bytes, this is equivalent to moving by one element in the column dimension and by two elements in the row dimension. The second row in the Numpy array starts at eight bytes (two elements) from the first entry (i.e., `2`) and then proceeds by four bytes (by one element) in the column dimension (i.e, `2,3,4,5`). The important part is that memory is re-used in the resulting `7x4` Numpy array. The code below demonstrates this by reassigning elements in the original `x` array. The changes show up in the `y` array because they point at the same allocated memory.

```
>>> x[::2]=99 # assign every other value
>>> x
array([99,  1, 99,  3, 99,  5, 99,  7, 99,  9, 99, 11, 99, 13, 99, 15])
>>> y # the changes appear because y is a view
array([[99,  1, 99,  3],
       [99,  3, 99,  5],
       [99,  5, 99,  7],
       [99,  7, 99,  9],
       [99,  9, 99, 11],
       [99, 11, 99, 13],
       [99, 13, 99, 15]])
```

Bear in mind that `as_strided` does not check that you stay within memory block bounds. So, if the size of the target matrix is not filled by the available data, the remaining elements will come from whatever bytes are at that memory location. In other words, there is no default filling by zeros or other strategy that defends memory block bounds. One defense is to explicitly control the dimensions as in the following code,

```
>>> n = 8 # number of elements
>>> x = arange(n) # create array
>>> k = 5 # desired number of rows
>>> y = as_strided(x,(k,n-k+1),(x.itemsize,)*2)
>>> y
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5],
       [3, 4, 5, 6],
       [4, 5, 6, 7]])
```

Array element-wise operations

The usual pairwise arithmetic operations are element-wise in Numpy.

```
>>> x*3
array([[[ 0,  3,  6,  9],
        [12, 15, 18, 21],
        [24, 27, 30, 33]],
       ...
       [[36, 39, 42, 45],
        [48, 51, 54, 57],
        [60, 63, 66, 69]]])
>>> x/float(x.max())
array([[[ 0.        ,  0.04347826,  0.08695652,  0.13043478],
        [ 0.17391304,  0.2173913 ,  0.26086957,  0.30434783],
        [ 0.34782609,  0.39130435,  0.43478261,  0.47826087]],
       ...
       [[ 0.52173913,  0.56521739,  0.60869565,  0.65217391],
        [ 0.69565217,  0.73913043,  0.7826087 ,  0.82608696],
        [ 0.86956522,  0.91304348,  0.95652174,  1.        ]]])
>>> np.sin(_) * np.exp(-_)
array([[[ 0.        ,  0.04161529,  0.0796141 ,  0.11416005],
        [ 0.14541567,  0.17354184,  0.19869738,  0.22103863],
        [ 0.24071911,  0.25788917,  0.27269573,  0.28528203]],
       ...
       [[ 0.29578742,  0.30434719,  0.31109238,  0.31614969],
        [ 0.3196414 ,  0.32168525,  0.32239443,  0.32187752],
        [ 0.32023851,  0.31757678,  0.31398715,  0.30955988]]])
```

Unary Functions

Now that we know how to create and manipulate Numpy arrays, let's consider how to compute with other Numpy features. Unary functions (*ufuncs*) are Numpy functions that are optimized to compute Numpy arrays at the C-level (i.e., outside the Python interpreter). Let's compute the trigonometric sine,

```
>>> a = np.linspace(0,1,20)
>>> np.sin(a)
array([ 0.        ,  0.05260728,  0.10506887,  0.15723948,  0.20897462,
        0.26013102,  0.310567  ,  0.36014289,  0.40872137,  0.45616793,
        0.50235115,  0.54714315,  0.59041986,  0.63206143,  0.67195255,
        0.70998273,  0.74604665,  0.78004444,  0.81188195,  0.84147098])
```

Note that that Python has a built-in `math` module with its own sine function.

```
>>> from math import sin
>>> [sin(i) for i in a]
[0.0, 0.05260728333807213, 0.10506887376594912, 0.15723948186175024,
0.20897462406278547, 0.2601310228046501, 0.3105670033203749,
0.3601428860007191, 0.40872137322898616, 0.4561679296190457,
0.5023511546035125, 0.547143146340223, 0.5904198559291864,
0.6320614309590333, 0.6719525474315213, 0.7099827291448582,
0.7460466536513234, 0.7800444439418607, 0.8118819450498316,
0.8414709848078965]
```

Note that the output is a list, not a Numpy array, but in order to process all the elements of `a`, we had to use the looping semantics of the list comprehensions to compute the sine. This is because Python's `math` function only works one-at-a-time with each member of the array. Numpy's sine function does not need these extra semantics because the computing runs in the Numpy C-code outside of the Python interpreter.

Numpy Data Input/Output

Numpy makes it easy to move data in and out of files.

```
>>> x=np.loadtxt('sample1.txt')
>>> print x
[[  0.   0.]
 [  1.   1.]
 [  2.   4.]
 [  3.   9.]
 [  4.  16.]
 [  5.  25.]
 [  6.  36.]
 [  7.  49.]
 [  8.  64.]
 [  9.  81.]]

>>> x=np.loadtxt('sample1.txt',dtype=np.dtype('f8','i4'))
```

Numpy dtypes allow for formatting the columns in the written file.

Exercise using Numpy I/O

1. Using `np.savetxt()`, load sample1.txt and create a new file that has a third column which is the cube of the first column (e.g. [2,4,8]).

Linear algebra

Numpy has direct access to the battle-tested LAPACK/BLAS linear algebra code. The main entryway to linear algebra functions in Numpy is via the `linalg` submodule,

```
>>> np.linalg.eig(np.eye(3)) # runs underlying LAPACK/BLAS
(array([ 1.,  1.,  1.]), array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]]))

>>> np.eye(3)*np.arange(3) # does this work as expected?
array([[ 0.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  2.]])
```

To get matrix row-column products, you can use the `matrix` object,

```
>>> np.eye(3)*np.matrix(np.arange(3) ).T # row-column multiply,
matrix([[ 0.],
        [ 1.],
        [ 2.]])
```

More generally, you can use Numpy's `dot` product,

```
>>> a = np.eye(3)
>>> b = np.arange(3).T
>>> a.dot(b)
array([ 0.,  1.,  2.])
>>> b.dot(b)
5
```

The advantage of `dot` is that it works in arbitrary dimensions. This is handy for tensor-like contractions (see Numpy `tensordot` for more info).

Numpy Linear Algebra

1. Verify the following distributive property of determinants. That is, given two conformable matrices of your choosing, check that $det(A * B) = det(A) * det(B)$.

2. Create a square matrix and verify that the determinant of that matrix is equal to the product of the diagonals of the singular value matrix in the singular value decomposition.

Broadcasting

Broadcasting is incredibly powerful but hard to understand. Formally,

Quoted from the **Guide to NumPy** by Travis Oliphant:

1. All input arrays with ndim smaller than the input array of largest ndim have 1's pre-pended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input shapes in that dimension.
3. An input can be used in the calculation if it's shape in a particular dimension either matches the output shape or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the ufunc will simply not step along that dimension when otherwise needed (the stride will be 0 for that dimension).

An easier way to think about these rules is the following,

1. If the array shapes have different lengths, then left-pad the smaller shape with ones.
2. If any corresponding dimension does not match, make copies along the 1-dimension
3. If any corresponding dimension does not have a *1* in it, raise an error.

Hopefully, some examples will help. Consider these two arrays:

```
>>> from numpy import arange
>>> x = arange(3)
>>> y = arange(5)
```

and I want to compute the element wise product of these. I could use a loop like:

```
>>> out = []
>>> for i in x:
...    for j in y:
...       out.append(i*j)
>>> print out
[0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 2, 4, 6, 8]
```

If I additionally want to organize the output as a shaped array, I could do initialize out differently:

```
>>> out=array(out).reshape(len(x),-1) # -1 means infer the remaining dimension
>>> print out
[[0 0 0 0 0]
 [0 1 2 3 4]
 [0 2 4 6 8]]
```

I could use matrix algebra and do:

```
>>> from numpy import matrix
>>> out=matrix(x).T * y
>>> print out
[[0 0 0 0 0]
 [0 1 2 3 4]
 [0 2 4 6 8]]
```

But how can I generalize this to handle multiple dimensions? Let's consider adding a *singleton* dimension to `y` as

```
>>> print x[:,None].shape
(3, 1)
```

Note that we can use `np.newaxis` instead of `None` for readableness. Now, if we try this directly, broadcasting will handle the incompatible dimensions by making copies along the singleton dimension.

```
>>> print x[:,None]*y
[[0 0 0 0 0]
 [0 1 2 3 4]
 [0 2 4 6 8]]
```

and this works with more complicated expressions:

```
>>> from numpy import cos
>>> print x[:,None]*y + cos(x[:,None]+y)
[[ 1.          0.54030231 -0.41614684 -0.9899925  -0.65364362]
 [ 0.54030231  0.58385316  1.0100075   2.34635638  4.28366219]
 [-0.41614684  1.0100075   3.34635638  6.28366219  8.96017029]]
```

But, what if I don't like the shape of the resulting array?

```
>>> print x*y[:,None] # change the placement of the singleton dimension
[[0 0 0]
 [0 1 2]
 [0 2 4]
 [0 3 6]
 [0 4 8]]
```

Now, let's consider a bigger example

```
>>> X = arange(2*4).reshape(2,4)
>>> Y = arange(3*5).reshape(3,5)
```

and I want to no element wise multiply these two together. The result will be a *2 x 4 x 3 x 5* multidimensional matrix.

```
>>> print X[:,:,None,None] * Y
[[[[ 0  0  0  0  0]
   [ 0  0  0  0  0]
   [ 0  0  0  0  0]]

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

  [[ 0  2  4  6  8]
   [10 12 14 16 18]
   [20 22 24 26 28]]

  [[ 0  3  6  9 12]
   [15 18 21 24 27]
   [30 33 36 39 42]]]


 [[[ 0  4  8 12 16]
   [20 24 28 32 36]
   [40 44 48 52 56]]

  [[ 0  5 10 15 20]
   [25 30 35 40 45]
   [50 55 60 65 70]]
```

```
 [[ 0  6 12 18 24]
  [30 36 42 48 54]
  [60 66 72 78 84]]

 [[ 0  7 14 21 28]
  [35 42 49 56 63]
  [70 77 84 91 98]]]]
```

```
>>> print X[0][0] * Y # 1st array element
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
>>> print X[0][1] * Y # 2nd array element
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
>>> print X[0][2] * Y # 3rd array element
[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]]
```

```
>>> (X[:,:,None,None] * Y).sum(axis=3) # sum along 4th dimension
array([[[  0,   0,   0],
        [ 10,  35,  60],
        [ 20,  70, 120],
        [ 30, 105, 180]],

       [[ 40, 140, 240],
        [ 50, 175, 300],
        [ 60, 210, 360],
        [ 70, 245, 420]]])
```

Exercise: Abbreviating Loops with Broadcasting

Using broadcasting, compute the number of ways a quarter (i.e., 25 cents) can be split into pennies, nickels, and dimes.

Hint: You can do this using loops as

```
n=0 # start counter
for n_d in range(0,3): # at most 2 dimes
   for n_n in range(0,6): # at most 5 nickels
      for n_p in range(0,26): # at most 25 pennies
         value = n_d*10+n_n*5+n_p
         if value == 25:
            print 'dimes=%d, nickels=%d, pennies=%d'%(n_d,n_n,n_p)
            n+=1
```

- Can you find a better way using broadcasting?
- Can you do it in one-line?
- What is the supportability argument against this?

Exercise: Nearest neighbors via broadcasting

Using broadcasting, can you compute the nearest neighbors to the following two-dimensional array,

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

For a given row, your answer should be the indices of the other row that is closest to it (other than itself) using the square of the Euclidean distance metric

$$d_{i,j} = x_{i,j}^2 + y_{i,j}^2$$

The answer for this exercise is the following,

```
array([1, 0, 1, 2])
```

So, the point *(0,1)* in the 0th row is closest to *(2,3)* which is 1st row. The 1st row is closest to the 0th row, the 2nd row is closest to the 1st row, and the 3rd row is closest to the 2nd row.

What if you had a three-dimensional array?

Masked arrays

Numpy also allows for masking sections of Numpy arrays. This is very popular in image processing.

```
>>> x = np.array([2, 1, 3, np.nan, 5, 2, 3, np.nan])
>>> x
Out[5]: array([  2.,   1.,   3.,  nan,   5.,   2.,   3.,  nan])
>>> np.mean(x)
nan
>>> m = np.ma.masked_array(x, np.isnan(x))
>>> m
masked_array(data = [2.0 1.0 3.0 -- 5.0 2.0 3.0 --],
             mask = [False False False  True False False False  True],
       fill_value = 1e+20)
...
>>> np.mean(m)
2.6666666666666665
>>> m.shape
(8,)
>>> x.shape
(8,)
>>> m.fill_value=9999
>>> m.filled()
array([  2.00000000e+00,   1.00000000e+00,   3.00000000e+00,
         9.99900000e+03,   5.00000000e+00,   2.00000000e+00,
         3.00000000e+00,   9.99900000e+03])
```

Looking Back: Making Numpy Arrays from Custom Objects

To make custom objects compatible with Numpy arrays, we have to embue the object with the `__array__` method.

```
from numpy import arange
class Foo():
      def __init__(self): # note the double underscores
          self.size = 10
      def __array__(self): # produces numpy array
          return arange(self.size)

>>> np.array(Foo())
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Exercises

Exercise: Fibonacci numbers

1. The Fibonacci numbers are defined by the following recursion:

   $$F_n = F_{n-2} + F_{n-1}$$

   with initial values $F_1 = F_2 = 1$. Using Numpy arrays, compute the first ten Fibonacci numbers, `[1,1,2,3,5,8,13,21,34,55,89]`. Hint: Use the closed form for the Fibonacci sequence.

2. Given the following two-dimensional array, set the diagonal element to the average of the columns in each respective row not counting the 0 element in each row.

```
[[0,   3, 14, 13, 12],
 [13,  0,  8,  5, 11],
 [11, 11,  0, 12, 10],
 [11, 12,  1,  0, 10],
 [13, 12, 11,  4,  0]]
```

Your output should be,

```
[[ 10.5 ,   3.  ,  14.  ,  13.  ,  12.  ],
 [ 13.  ,   9.25,   8.  ,   5.  ,  11.  ],
 [ 11.  ,  11.  ,  11.  ,  12.  ,  10.  ],
 [ 11.  ,  12.  ,   1.  ,   8.5 ,  10.  ],
 [ 13.  ,  12.  ,  11.  ,   4.  ,  10.  ]]
```

Floating Point Numbers

There are precision limitations when representing floating point numbers on a computer with finite memory. For example, the following shows these limitations when adding two simple numbers,

```
>>> 0.1 + 0.2
0.30000000000000004
```

So, then, why is the output not `0.3`? The issue is the floating point representation of the two numbers and the algorithm that adds them. To represent an integer in binary, we just write it out in powers of `2`. For example, $230 = (11100110)_2$. Python can do this conversion using string formatting,

```
>>> print '{0:b}'.format(230)
```

To add integers, we just add up the corresponding bits and fit them into the allowable number of bits. Unless there is an overflow (the results cannot be represented with that number of bits), then there is no problem. Representing floating point is trickier because we have to represent these numbers as binary fractions. The IEEE 754 standard requires that floating-point numbers be represented as $\pm C \times 2^E$ where `C` is the significand (*mantissa*) and `E` is the exponent.

To represent a regular decimal fraction as binary fraction, we need to compute the expansion of the fraction in the following form $a_1/2 + a_2/2^2 + a_3/2^3 \ldots$ In other words, we need to find the a_i coefficients. We can do this using the same process we would use for a decimal fraction: just keep dividing by the fractional powers of $1/2$ and keep track of the whole and fractional parts. Python's `divmod` function can do most of the work for this. For example, to represent `0.125` as a binary fraction,

```
>>> a = 0.125
>>> divmod(a*2,1)
(0.0, 0.25)
```

The first item in the tuple is the quotient and the other is the remainder. If the quotient was greater than `1`, then the corresponding a_i term is one and is zero otherwise. For this example, we have $a_1 = 0$. To get the next term in the expansion, we just keep multiplying by `2` which moves us rightward along the expansion to a_{i+1} and so on. Then,

```
>>> a = 0.125
>>> q,a = divmod(a*2,1)
>>> print q,a
0.0 0.25
>>> q,a = divmod(a*2,1)
>>> print q,a
0.0 0.5
```

```
>>> q,a = divmod(a*2,1)
>>> print q,a
1.0 0.0
```

The algorithm stops when the remainder term is zero. Thus, we have that $0.125 = (0.001)_2$. The specification requires that the leading term in the expansion be one. Thus, we have $0.125 = (1.000) \times 2^{-3}$. This means the significand is `1` and the exponent is `-3`.

Now, let's get back to our main problem `0.1+0.2` by developing the representation `0.1` by coding up the individual steps above.

```
a = 0.1
bits = []
while a>0:
   q,a = divmod(a*2,1)
   bits.append(q)

>>> print ''.join(['%d'%i for i in bits])
0001100110011001100110011001100110011001100110011001101
```

Note that the representation has an infinitely repeating pattern. This means that we have $(1.\overline{1001})_2 \times 2^{-4}$. The IEEE standard does not have a way to represent infinitely repeating sequences. Nonetheless, we can compute this,

$$\sum_{n=1}^{\infty} \frac{1}{2^{4n-3}} + \frac{1}{2^{4n}} = \frac{3}{5}$$

Thus, $0.1 = 1.6 \times 2^{-4}$.

Per the IEEE 754 standard, for `float` type, we have 24 bits for the significand and 23 bits for the fractional part. Because we cannot represent the infinitely repeating sequence, we have to round off at 23 bits, `10011001100110011001101`. Thus, whereas the significand's representation used to be `1.6`, with this rounding, it is Now

```
print 1+sum([ int(i)/(2**n) for n,i in enumerate('10011001100110011001101',1) ])
1.600000023842
```

Thus, we now have $0.1 \approx 1.600000023842 \times 2^{-4} = 0.100000001490125$. Using the same procedure for `0.2`, we obtain the following:

```
print (1+sum([ int(i)/(2**n) for n,i in enumerate('10011001100110011001110',1)]))
1.60000014305
```

Thus, we have $0.2 \approx 1.60000014305 \times 2^{-3} = 0.20000001788125$.

To add `0.1+0.2` in binary, we must adjust the exponents until they match the higher of the two. Thus,

```
 0.1100110011001100110011 0
+1.1001100110011001100111 0
 -------------------------
10.0110011001100110011010 0
```

Now, the sum has to be scaled back to fit into the significand's available bits so the result is `1.00110011001100110011010` with exponent `-2`. Computing this in the usual way as shown below gives the result.

```
print (1+sum([int(i)/(2**n) for n,i in enumerate('00110011001100110110100',1)]))/2**2
0.300000011921
```

which matches what we get with `numpy`

```
import numpy as np
print '%0.12f'%(np.float32(0.1) + np.float32(0.2))
0.300000011921
```

The entire process proceeds the same for 64-bit floats. Python has a `fractions` and `decimal` modules that allow more exact number representations. The `decimal` module is particularly important for certain financial computations.

Roundoff Error

Let's consider the example of adding `100,000,000` and `10` in 32-bit floating point.

```
>>> print '{0:b}'.format(100000000)
10111110101111000010000000
```

This means that $100,000,000 = (1.01111101011110000100000000)_2 \times 2^{26}$. Likewise, $10 = (1.010)_2 \times 2^3$. To add these we have to make the exponents match as in the following,

```
 1.01111101011110000100000000
+0.00000000000000000000001010
------------------------------
 1.01111101011110000100001010
```

Now, we have to round off because we only have 23 bits to the right of the decimal point and obtain `1.01111101011110000100000`, thus losing the trailing `10` bits. This effectively makes the decimal $10 = (1010)_2$ we started out with become $8 = (1000)_2$. Thus, using Numpy again,

```
>>> print format(np.float32(100000000) + np.float32(10),'10.3f')
100000008.000
```

The problem here is that the order of magnitude between the two numbers was so great that it resulted in loss in the significand's bits as the smaller number was right-shifted. When summing numbers like these, the Kahan Summation algorithm (see `math.fsum()`) can effectively manage these round-off errors.

```
>>> import math
>>> math.fsum([np.float32(100000000),np.float32(10)])
100000010.0
```

Cancellation Error

Cancellation error (loss of significance) results when two nearly equal floating-point numbers are subtracted. Let's consider subtracting `0.1111112` and `0.1111111`. As binary fractions, we have the following,

```
 1.11000111000111001000101 E-4
-1.11000111000111000110111 E-4
-------------------------------
 0.00000000000000000011100
```

As a binary fraction, this is `1.11` with exponent `-23` or $(1.75)_{10} \times 2^{-23} \approx 0.00000010430812836$. In Numpy, this loss of precision is shown in the following,

```
>>> print format(np.float32(0.1111112)-np.float32(0.1111111),'1.17f')
0.00000010430812836
```

To sum up, when using floating point, you must check for approximate equality using something like Numpy `allclose` instead of the usual Python equality sign. This enforces error bounds instead of strict equality. Whenever practicable, use fixed scaling to employ integer values instead of decimal fractions. Double precision 64-bit floating point numbers are much better than single precision and, while not eliminating these problems, effectively kicks the can down the road for all but the strictest precision requirements. The Kahan algorithm is effective for summing floating point numbers across very large data without accruing round-off errors. To minimize cancellation errors, re-factor the calculation to avoid subtracting two nearly equal numbers.

Additional Resources

- Numpy cheatsheet
- From Python to Numpy
- A Quick Introduction to the NumPy Library
- How to do Descriptive Statistics in Python using Numpy
- NumPy 1.5 Beginner's Guide: Ivan Idris
- How to do Descriptives Statistics in Python using Numpy - Erik Marsja

Numpy videos

- Diving into NumPy Code, SciPy2013 Tutorial, Part 1 of 4 - YouTube
- numexpr - Fast numerical array expression evaluator for Python and NumPy.
- Theano: A package for efficient computation in Python | GPU Computing
- PyCUDA | Andreas Klockner's web page

C/FORTRAN Integration

- Simplified Wrapper and Interface Generator
- Cython: C-Extensions for Python
- F2py - Fortran to Python converter
- Cython: Speed up Python and NumPy, Pythonize C, C++, and Fortran, SciPy2013 Tutorial, Part 1 of 4 - YouTube
- Advanced Cython Recorded Webinar: Typed Memoryviews | Enthought[s]

MATLAB Integration and Tips

- Numpy for Matlab Users