Chap 1 - Introduction to NumPy Library

March 23, 2018

To import the NumPy library, we mostly do like this.

```
In [1]: import numpy as np
```

At this point, we can use the library by typing `np`.

0.0.1 Data Types and Shapes

The most common way to work with numbers in NumPy is through ndarray object. They are similar to Python lists but can have *any number of dimensions*. We can use `ndarrays` to represent: scalars, vectors, matrices or tensors.

```
In [6]: ndarray1 = np.ndarray(shape=(2,3), dtype=int, order='C') #'C' indicates row-major
        print(ndarray1)
        print('shape: {}'.format(ndarray1.shape))
        print('size: {}'.format(ndarray1.size))
        # Reshape to (1,6)
        ndarray1.shape = 1,6
        print('Reshape to (1,6):\n{}'.format(ndarray1))
```

```
[[140704807578552 140704807578552 140704749419856]
 [140704811679280        10919456        10919488]]
shape: (2, 3)
size: 6
Reshape to (1,6):
[[140704807578552 140704807578552 140704749419856 140704811679280
         10919456        10919488]]
```

```
In [8]: ndarray2 = np.ndarray(shape=(2,3), dtype=int, order='F') # 'F' indicates column-major
        print(ndarray2)
        print('shape: {}'.format(ndarray2.shape))
        print('size: {}'.format(ndarray2.size))
```

```
[[   140704807578536  72059256190493952 316659363194536067]
 [  1517326119993346           43035040 460494162802771556]]
shape: (2, 3)
size: 6
```

0.0.2 Scalars in NumPy

There are more variety for Sclars in NumPy. For example, instead of Python's `int`, we have access to types like `uint8`, `int8`, `uint16`, `int16`, and so on.

Note that when we create a NumPy array, we can specify the type but **every item in the array must have the same type**. That is NumPy arrays are more like C arrays than Python lists.

To create a NumPy array that holds a scalar, we can do by passing a value to NumPy's `array` function:

```
In [14]: s1 = np.array(10)
         s2 = np.array(3.5)
         print('Type of s1: {}'.format(type(s1)))
         print('Type of s2: {}'.format(type(s2)))
         print('Shape of s1: {}'.format(s1.shape)) # 0 dimension
         print('Shape of s2: {}'.format(s2.shape)) # 0 dimension
```

```
Type of s1: <class 'numpy.ndarray'>
Type of s2: <class 'numpy.ndarray'>
Shape of s1: ()
Shape of s2: ()
```

We can still perform math between `ndarrays`, NumPy scalars, and normal Python scalars.

They say even though scalars are insdie arrays, we can still use them like a normal scalar. Let's try

```
In [17]: x1 = s1 + 20
         x2 = s2 + 2.5
         x3 = s1 + s2
         print('x1 = ' + str(x1))
         print('x2 = ' + str(x2))
         print('x3 = ' + str(x3))
         print('Type of x1 :', type(x1)) # Notice, the type is NumPy type
         print('Type of x2 :', type(x2)) # Notice, the type is NumPy type
         print('Type of x3 :', type(x3)) # Notice, the type is NumPy type
```

```
x1 = 30
x2 = 6.0
x3 = 13.5
Type of x1 : <class 'numpy.int64'>
Type of x2 : <class 'numpy.float64'>
Type of x3 : <class 'numpy.float64'>
```

By the way, even scalar types support most of the array functions. Here, `x1` is a scalar of type `numpy.int64`. Try calling `x1.shape`, in which shape is a property of arrays.

```
In [18]: print('Shape of x1 :', x1.shape) # 0 dimension
```

```
Shape of x1 : ()
```

0.0.3 Vectors

To create a vector, pass a Python list to the `array` function:

```
In [19]: v = np.array([1,2,3])
         print('v =', v)
         print('Shape of v:', v.shape)
```

```
v = [1 2 3]
Shape of v: (3,)
```

A vector's `shape` attributue will return a single number representing the vector's 1-D length. We can access an element within the vector using indices, for example:

```
In [20]: print(v[0]) # 1
         print(v[1]) # 2
         print(v[2]) # 3
         print(v[-1]) # 3, the last element
         print(v[-2])
```

```
1
2
3
3
2
```

NumPy also supports advanced indexing technqiues. This is called slicing

```
In [22]: v = np.array([1,2,3,4,5,6])
         print(v)
         print(v[1:]) # access the items from index 1 onwards (0-based)
         print(v[3:]) # access the items from index 3 onwards
```

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

```
In [28]: v = np.array([0,1,2,3,4,5,6,7,8,9])
         print(v)
         print(v[0:9:2]) # start from index 0 to 9 with step = 2
         print(v[1:9:2]) # start from index 1 to 9 with step = 2
         print(v[0:-1:2])
```

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

0.0.4 Matrices

We create matrices using NumPy's `array` function but, instead of passing in a list, we need to pass a list of lists, when each list represents a row.

```
In [31]: m1 = np.array([[1,2],[3,4],[5,6]])
         m2 = np.array([[1],[2],[3]])
         print('m1:',m1)
         print('m1 shape:', m1.shape)
         print('m2:',m2)
         print('m2 shape:', m2.shape)
```

```
m1: [[1 2]
 [3 4]
 [5 6]]
m1 shape: (3, 2)
m2: [[1]
 [2]
 [3]]
m2 shape: (3, 1)
```

To access elements of matrices, we use two index values: row index and column index.

```
In [32]: print('m1[0][1]:', m1[0][1])
         print('m1[2][0]:', m1[2][0])
```

```
m1[0][1]: 2
m1[2][0]: 5
```

0.0.5 Tensors

Tensors are just like vectors and matrices but they can have more dimensions.

```
In [38]: # 3D tensor : 4 matrices of 3 rows x 2 columns
         t1 = np.array([[[1,2],[3,4],[5,6]],\
                        [[11,12],[13,14],[15,16]],\
                        [[111,112],[113,114],[115,116]],\
                        [[1111,1112],[1113,1114],[1115,1116]]])
         print('t1:', t1)
         print(t1.shape)
```

```
t1: [[[   1    2]
  [   3    4]
  [   5    6]]

 [[  11   12]
  [  13   14]
  [  15   16]]
```

```
 [[ 111  112]
  [ 113  114]
  [ 115  116]]

 [[1111 1112]
  [1113 1114]
  [1115 1116]]]
(4, 3, 2)
```

```
In [40]: # 4D tensor : 2 x (4 matrices of 3 rows x 2 columns)
         t2 = np.array([[[[1,2],[3,4],[5,6]],\
                         [[11,12],[13,14],[15,16]],\
                         [[111,112],[113,114],[115,116]],\
                         [[1111,1112],[1113,1114],[1115,1116]]],\
                        [[[51,52],[53,54],[55,56]],\
                         [[51,12],[13,14],[15,16]],\
                         [[511,112],[113,114],[115,116]],\
                         [[5111,1112],[1113,1114],[1115,1116]]]
                        ])
         print('t2:', t2)
         print(t2.shape)
```

```
t2: [[[[   1    2]
   [   3    4]
   [   5    6]]

  [[  11   12]
   [  13   14]
   [  15   16]]

  [[ 111  112]
   [ 113  114]
   [ 115  116]]

  [[1111 1112]
   [1113 1114]
   [1115 1116]]]


 [[[  51   52]
   [  53   54]
   [  55   56]]

  [[  51   12]
   [  13   14]
   [  15   16]]
```

```
  [[ 511  112]
   [ 113  114]
   [ 115  116]]

  [[5111 1112]
   [1113 1114]
   [1115 1116]]]]
(2, 4, 3, 2)
```

```
In [41]: # 4D tensor : (3 matrices of 2 rows x 2 columns)
         t3 = np.array([[[1,2],[1,2]],\
                        [[11,22],[11,22]],\
                        [[111,222],[111,222]]
                       ])
         print('t3:', t3)
         print(t3.shape)
```

```
t3: [[[  1   2]
  [  1   2]]

 [[ 11  22]
  [ 11  22]]

 [[111 222]
  [111 222]]]
(3, 2, 2)
```

```
In [50]: # 4D tensor : 2 x(2 matrices of 1 rows x 3 columns)
         # 1 rows x 3 cols
         t1 = np.array([1,1,1])
         print('t1:', t1)
         print('t1 shape:', t1.shape)
         # 2 matrices of 1 rows x 3 colums
         # t2 = np.array([[t1],[t1]])
         t2 = np.array([[t1],[t1]])
         print('t2:', t2)
         print('t2 shape: ', t2.shape)
         t2_ = np.array([[[1,1,1]],[[1,1,1]]]) # equivalent to t2
         print('t2_:', t2_)
         print('t2_ shape: ', t2_.shape)
         # 2 of (2 matrices of 1 rows x 3 columns)
         t3 = np.array([
                       [[[1,1,1]],[[2,2,2]]],\
                       [[[3,3,3]],[[4,4,4]]]
                       ])
```

```
        print('t3:', t3)
        print('t3 shape: ', t3.shape)
```

```
t1: [1 1 1]
t1 shape: (3,)
t2: [[[1 1 1]]

 [[1 1 1]]]
t2 shape:  (2, 1, 3)
t2_: [[[1 1 1]]

 [[1 1 1]]]
t2_ shape:  (2, 1, 3)
t3: [[[[1 1 1]]

  [[2 2 2]]]


 [[[3 3 3]]

  [[4 4 4]]]]
t3 shape:  (2, 2, 1, 3)
```

0.0.6 Changing Shapes

Sometimes, we'll need to change the shape of our data without changing its content.

```
In [51]: v = np.array([1,2,3,4])
         print(v.shape) # this is a vector
```

```
(4,)
```

What if we want a $1x4$ matrix or a $4x1$ matrix ? We can accomplishe that with a reshape function.

```
In [52]: x1 = v.reshape(1,4)
         print('x1:', x1)
         print(x1.shape)
```

```
x1: [[1 2 3 4]]
(1, 4)
```

```
In [53]: x2 = v.reshape(4,1)
         print('x2:', x2)
         print(x2.shape)
```

```
x2: [[1]
 [2]
 [3]
 [4]]
(4, 1)
```

We could also use a slicing syntax instead of **reshape**.

```
In [54]: x1 = v[None,:] # keep the number of columns
         print('x1:', x1)
         print(x1.shape)
         x2 = v[:,None] # keep the number of rows
         print('x2:', x2)
         print(x2.shape)
```

```
x1: [[1 2 3 4]]
(1, 4)
x2: [[1]
 [2]
 [3]
 [4]]
(4, 1)
```