Numpy Crash Course

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

Creating Array of Zeros

In [2]:
```
a = np.zeros((2, 2))   # Create an array of all zeros of specified shape
print(a)
```

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

Creating Array of Ones

In [3]:
```
a = np.ones((2, 2))    # Create an array of all ones
print(a)
```

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

Creating an Array of Constant Values

Here we chose the constant value to be `7`

In [4]:
```
b = np.full((2, 2), 7)  # Create a constant array
print(b)
```

```
[[7 7]
 [7 7]]
```

Creating an Identity Matrix

In [5]:
```
c = np.eye(2)   # Create a 2 x 2 identity matrix
print(c)
```

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

Creating an Array of Random Values

In [6]:
```
d = np.random.randint(5,10, size=(2, 2))  # Create a 2x2 array filled with randor
print(d)
```

```
[[9 9]
 [5 8]]
```

Reshaping an Array

In [7]:
```
nums = np.arange(16)
print(nums)
print(nums.shape)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
(16,)
```

Now lets reshape that 16, to a 4x4 array

In [8]:
```
nums = nums.reshape((4, 4))
print('Reshaped:\n', nums)
print(nums.shape)
```

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

Using -1

The -1 in reshape corresponds to an unknown dimension that numpy will figure out based on all other dimensions and the array size. We Can only specify one unknown dimension. For example, sometimes we might have an unknown number of data points, and so we can use -1 instead without worrying about the true number.

In [9]:
```
nums = nums.reshape((4,-1 ))
print('Reshaped with -1:\n', nums)
print(nums.shape)
```

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

NumPy supports an object-oriented paradigm, such that ndarray has a number of methods and attributes, with functions similar to ones in the outermost NumPy namespace.
For example, we can do both:

In [10]:
```
nums = np.arange(8)
print(nums.min())
print(np.min(nums))
```

```
0
0
```

Flatten vs Ravel

The primary functional difference is that flatten() is a method of an ndarray object and hence can only be called for true numpy arrays. In contrast ravel() is a library-level function and hence can be called on any object that can successfully be parsed. For example ravel() will work on a list of ndarrays, while flatten (obviously) won't

In [11]:
```
print (b)
```

```
[[7 7]
 [7 7]]
```

In [12]:
```
b.flatten()
```

Out[12]:
```
array([7, 7, 7, 7])
```

In [13]:
```
# flattening the array ...used in computer vision a lot..
b.ravel()
```

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

Array Operations/Math

NumPy supports many elementwise operations:

In [14]:
```
x = np.array([[1, 2],
              [3, 4]], dtype=np.float64)
y = np.array([[5, 6],
              [7, 8]], dtype=np.float64)
```

Addition

In [16]:
```
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

Subtraction

In [17]:
```
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

Multiplication

In [18]:
```
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

Division

How do we elementwise divide between two arrays?

In [19]:
```
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

Note: This is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects

Numpy Functions

There are many useful functions built into NumPy, and often we're able to express them across specific axes of the ndarray:

In [20]:
```
x = np.array([[1, 2, 3],
              [4, 5, 6]])

print(np.sum(x))          # Compute sum of all elements
print(np.sum(x, axis=0))  # Compute sum of each row
print(np.sum(x, axis=1))  # Compute sum of each col
```

```
21
[5 7 9]
[ 6 15]
```

Note: `Axis 0 is row and Axis 1 is column`

np.Max

In [21]:
```
print(np.max(x, axis=1))  # Compute max of each row
```

```
[3 6]
```

Argmax

How can we compute the index of the max value of each row? Useful, to say, find the class that corresponds to the maximum score for an input image.

In [23]:
```
x = np.array([[1, 7, 3],
              [4, 5, 6]])

print(np.argmax(x, axis=0)) # Compute index of max of each row
```

```
[1 0 1]
```

Computing on a specific Axis

In [24]:
```
x = np.array([[1,2],[3,4]])
print("array: ",x)
print("-----")
print ("sum of all elements : ",np.sum(x))  # Compute sum of all elements; prints
print ("sum of rows : ", np.sum(x, axis=0))  # Compute sum of each rows; prints
print ("sum of cols : " ,np.sum(x, axis=1))  # Compute sum of each cols; prints
```

```
array:  [[1 2]
 [3 4]]
-----
sum of all elements :  10
sum of rows :  [4 6]
sum of cols :  [3 7]
```

Slicing

Numpy slicing is pretty similar to python list slicing

In [27]:
```
b[0:2,0]  # from row 0 and 1 take column 1st
```

Out[27]:
```
array([7, 7])
```

In [28]:
```
x[:,1:2]  # gives all rows of col 2
```

Out[28]:
```
array([[2],
       [4]])
```

Reversing an axis

In [29]:
```
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

# The following reverses the first row and prints it out
print('Reversing the first row (a[0, ::-1]) :\n', a[0, ::-1])
```

```
Reversing the first row (a[0, ::-1]) :
 [4 3 2 1]
```