# Numpy Tutorial

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


# This import is needed so that we can display full output in Jupyter, not onl
y the last result.
# Importing modules is explained later in this tutorial.
# For the moment just execute this cell.

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

# Arrays

The main entity is the "array". They are also called tensors.

Arrays have a shape, e.g. 3x4, which is represented by a tuple, e.g. (3,4).

In [2]:
```
a = np.array([1, 2, 3])  # Create a rank 1 array
print (a.shape)            # Prints "(3,)"
print (a[0], a[1], a[2])   # Prints "1 2 3"

a[0] = 5                 # Change an element of the array
print (a)                  # Prints "[5, 2, 3]"
```

```
(3,)
1 2 3
[5 2 3]
```

In [3]:
```
b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print (b.shape)                     # Prints "(2, 3)"
print (b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
(2, 3)
1 2 4
```

# Array Math

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

In [5]:
```
# Elementwise sum
z = x + y
print ("x+y = \n", z, "\n")

# Elementwise difference
z = x - y
print ("x-y = \n", z, "\n")

# Elementwise product
z = x * y
print ("x*y = \n", z, "\n")
```

```
x+y =
 [[ 6  8]
 [10 12]]

x-y =
 [[-4 -4]
 [-4 -4]]

x*y =
 [[ 5 12]
 [21 32]]
```

In [6]:
```
# Matrix / matrix product;
# There are three different ways -- same result
z = x.dot(y)
print ("x.dot(y) = \n", z, "\n")

z = np.dot(x, y)
print ("np.dot(x, y) = \n", z, "\n")

z = x @ y     # Only in Python 3
print ("x @ y = \n", z, "\n")
```

```
x.dot(y) =
 [[19 22]
 [43 50]]

np.dot(x, y) =
 [[19 22]
 [43 50]]

x @ y =
 [[19 22]
 [43 50]]
```

In [7]:
```
# Matrix transpose
z = x.T
print ("x.T = \n", z, "\n")
```

```
x.T =
 [[1 3]
 [2 4]]
```

In [8]:
```
# Multiplying a matrix with a number
z = x * 2
print ("x*2 = \n", z, "\n")
```

```
x*2 =
 [[2 4]
 [6 8]]
```

In [9]:
```
# Create a 3x4 array of zeros
x = np.zeros((3,4))
print(x)
```

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

In [10]:
```
# Create a 3x4 array of ones
x = np.ones((3,4))
print(x)
```

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

In [11]:
```
# Create an array of the given shape and populate it
# with random samples from a uniform distribution over [0, 1).

x = np.random.rand(3,4)
print(x)
```

```
[[ 0.07866137  0.23481423  0.26263778  0.70667897]
 [ 0.06512371  0.27459599  0.12016634  0.70678972]
 [ 0.03466456  0.52554067  0.21104205  0.75129968]]
```

In [12]:
```
# Create an array of the given shape and populate it
# with random samples from "standard normal" distribution
# (mean = 0, var = 1).

x = np.random.randn(3,4)
print(x)
```

```
[[ 0.41405373 -0.16982008 -0.06334137  0.0408446 ]
 [ 0.18693493  0.6919127   0.76373462  1.03858241]
 [-0.79806188 -0.35955393  1.11498648 -1.05110482]]
```

# Reshaping Arrays

In [13]:
```
print("x = \n",x)

z = np.reshape(x, (4,3))

print("z = \n", z)
```

```
x =
 [[ 0.41405373 -0.16982008 -0.06334137  0.0408446 ]
 [ 0.18693493  0.6919127   0.76373462  1.03858241]
 [-0.79806188 -0.35955393  1.11498648 -1.05110482]]
z =
 [[ 0.41405373 -0.16982008 -0.06334137]
 [ 0.0408446   0.18693493  0.6919127 ]
 [ 0.76373462  1.03858241 -0.79806188]
 [-0.35955393  1.11498648 -1.05110482]]
```

In [14]:
```
# We will often want to reshape 2D images to nx1 or 1xn arrays.

z = np.reshape(x,(x.shape[0]*x.shape[1],1))

print("z = \n", z)
```

```
z =
 [[ 0.41405373]
 [-0.16982008]
 [-0.06334137]
 [ 0.0408446 ]
 [ 0.18693493]
 [ 0.6919127 ]
 [ 0.76373462]
 [ 1.03858241]
 [-0.79806188]
 [-0.35955393]
 [ 1.11498648]
 [-1.05110482]]
```

In [15]:
```
# If we also call x.reshape(...); same as np.reshape(x,...)

z = x.reshape((x.shape[0]*x.shape[1],1))

print("z = \n", z)
```

```
z =
 [[ 0.41405373]
 [-0.16982008]
 [-0.06334137]
 [ 0.0408446 ]
 [ 0.18693493]
 [ 0.6919127 ]
 [ 0.76373462]
 [ 1.03858241]
 [-0.79806188]
 [-0.35955393]
 [ 1.11498648]
 [-1.05110482]]
```

# Array Slicing

Array slicing is similar to list slicing, but here we need to do slicing in each dimension.

In [16]:
```
# Let's create an array
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [17]:
```
# Slice in the 1st dimension from 0 to 2 (not including 2)
# Slice in the 2nd dimension from 1 to 3 (not including 3)
b = a[0:2, 1:3]

print(b)
```

```
[[2 3]
 [6 7]]
```

In [18]:
```
# The previous example can also be written as follows.
# We don't need to specify 0.
b = a[:2, 1:3]

print(b)
```

```
[[2 3]
 [6 7]]
```

In [19]:
```
# Here is another example
b = a[:2, 1:4]

print(b)
```

```
[[2 3 4]
 [6 7 8]]
```

In [20]:
```
# The previous example can also be written as follows.
# We don't need to specify 4.
b = a[:2, 1:]

print(b)
```

```
[[2 3 4]
 [6 7 8]]
```

In [21]:
```
# Negative indexes are used to count backwards from the end of a range.
# Suppose we want all the rows of an array, and all the columns, except the la
st column.

# Let's recall a
print("a = \n", a)

# Now let's take the slice we want
b = a[:, :-1]

print("b = \n", b)
```

```
a = 
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
b = 
 [[ 1  2  3]
 [ 5  6  7]
 [ 9 10 11]]
```

# Numpy Math Functions

In [22]:
```
# Let's create a 1x3 array
a = np.array([[1,2,3,4]])

# compute e^x for each number in the array
print("np.exp(a) = \n", np.exp(a))

# compute the log (base e) of each number in the array
print("np.log(a) = \n", np.log(a))

# See, numpy functions, such as exp and log,
# take arrays as input and produce arrays as output.
# They compute the function on each element of the input array.
```

```
np.exp(a) =
 [[  2.71828183   7.3890561   20.08553692  54.59815003]]
np.log(a) =
 [[ 0.          0.69314718  1.09861229  1.38629436]]
```

### The Sigmoid Function

An important function in neural networks is the Sigmoid function, sometimes known as the logistic function

$$sigmoid(x) = \frac{1}{1+e^{-x}}.$$

Let's implement it using numpy.

In [23]:
```
def sigmoid(x):
    return 1/(1+np.exp(-x))

sigmoid(a)
```

Out[23]:
```
array([[ 0.73105858,  0.88079708,  0.95257413,  0.98201379]])
```

### Axis and keepdims

Some numpy functions, such as sum, avg, min, max, etc, take as input an array and return a number, e.g. the sum, average, min, max of the elements of the array. What if we want the sum, average, min, max of each row or each column? For this we use *axis* and *keepdims*. See examples below.

In [24]:
```
# Let's create an array
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

a

# Sum of all elements in the array
np.sum(a)

# Sum of elements in each column
np.sum(a, axis=0)

# Sum of elements in each row
np.sum(a, axis=1)

# The problem with the above is that the result is not a 2D array.
# If we need the result to be a 2D array, specify keepdims=True.

np.sum(a, keepdims=True)

np.sum(a, axis=0, keepdims=True)

np.sum(a, axis=1, keepdims=True)
```

Out[24]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[24]:
```
78
```

Out[24]:
```
array([15, 18, 21, 24])
```

Out[24]:
```
array([10, 26, 42])
```

Out[24]:
```
array([[78]])
```

Out[24]:
```
array([[15, 18, 21, 24]])
```

Out[24]:
```
array([[10],
       [26],
       [42]])
```

# Broadcasting

We can perform operations with arrays of different shapes. For example suppose we have an array

$$a = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

and would like to multiply it with

$$b = \begin{bmatrix} 1 \\ 5 \\ 9 \end{bmatrix}$$

If we say $a * b$, array $b$ will be first expanded (broadcast) to

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 5 & 5 & 5 & 5 \\ 9 & 9 & 9 & 9 \end{bmatrix}$$

then element-wise multiplication will be performed.

In [25]:
```
a

b = np.array([[1],[5],[9]])

b

a*b
```

Out[25]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[25]:
```
array([[1],
       [5],
       [9]])
```

Out[25]:
```
array([[  1,   2,   3,   4],
       [ 25,  30,  35,  40],
       [ 81,  90,  99, 108]])
```

# The Softmax Function -- Exercise

(Adapted from Andrew Ng's exercise in Coursera, deeplearning.ai)

**Exercise**: Implement a softmax function using numpy. Softmax is a normalizing function used when the algorithm needs to classify two or more classes.

**Instructions**:

- for $x \in \mathbb{R}^{1 \times n}$
  $$\text{softmax}(x) = \text{softmax}(\begin{bmatrix} x_1 && x_2 && \dots && x_n \end{bmatrix}) = \begin{bmatrix} \frac{e^{x_1}}{\sum_{j}e^{x_j}} && \frac{e^{x_2}}{\sum_{j}e^{x_j}} && \dots && \frac{e^{x_n}}{\sum_{j}e^{x_j}} \end{bmatrix}$$
- for a matrix $x \in \mathbb{R}^{m \times n}$
  $$\text{softmax}(x) = \text{softmax}\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} \frac{e^{x_{11}}}{\sum_{j}e^{x_{1j}}} & \frac{e^{x_{12}}}{\sum_{j}e^{x_{1j}}} & \frac{e^{x_{13}}}{\sum_{j}e^{x_{1j}}} & \dots \\ \frac{e^{x_{21}}}{\sum_{j}e^{x_{2j}}} & \frac{e^{x_{22}}}{\sum_{j}e^{x_{2j}}} & \frac{e^{x_{23}}}{\sum_{j}e^{x_{2j}}} & \dots \\ \vdots & \vdots & \vdots & \ddots \\ \frac{e^{x_{m1}}}{\sum_{j}e^{x_{mj}}} & \frac{e^{x_{m2}}}{\sum_{j}e^{x_{mj}}} & \frac{e^{x_{m3}}}{\sum_{j}e^{x_{mj}}} & \dots \end{bmatrix}$$
  $$= \begin{pmatrix} \text{softmax(first row of x)} \\ \text{softmax(second row of x)} \\ \dots \\ \text{softmax(last row of x)} \end{pmatrix}$$

In [26]:
```
def softmax(x):
    # Create an array x_exp by applying np.exp() element-wise to x.

    # Create an array x_sum that contains the sum of each row of x_exp.
    # Use np.sum(..., axis = 1, keepdims = True).

    # Compute softmax(x) by dividing x_exp by x_sum. It should automatically u
se numpy broadcasting.
    # Return this array.
    return None


# Let's test
x = np.array([
    [1, 2, 3, 1, 2],
    [9, 5, 1, 0 ,0]])
print("softmax(x) = " + str(softmax(x)))
```

```
softmax(x) = None
```