Numpy

> Python by itself doesn't have any built in data types for arrays or matrices. Any direct (matrix multiplication) or indirect (image manipulation) operations that require arrays are done with the numpy package.

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. Numpy is one of the most commonly used and best supported libraries in python and is the basis of all numerical tools in python (most tools that we'll address from now on require it).

Numpy is similar to matlab, as it is a fairly low level tool (many packages however - like pandas - feature higher level tools and are built on top of numpy).

Arrays

Arrays are the most common data types in data science. A numpy array is a grid of values, all of the **same type**, and is **indexed by a tuple of nonnegative integers**.

- The **rank** of an array is the number of dimensions it has.
- The **shape** of an array is a tuple giving the size (or length) of each dimension.

In mathematical terms:

- A 0D array, i.e a number, is called a **scalar**.
- A 1D array is called a **vector**.
- A 2D array is called a **matrix**.
- A 3D array is called a **tensor**.

1D Arrays

One dimensional arrays are a lot like lists. The main difference is that np.arrays allow only certain data types, while lists are a lot more versatile.

Let's dive right in. We'll first define the following array in numpy:

$$A = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

In [1]:
```
from __future__ import print_function  # for python 2-3 compatibility
import numpy as np  # because we use numpy often, it's more convenient to refer t

A = np.array([1, 2, 3])  # this is how we define and initialize a known array
```

But, what **type** of an object is *A*?

In [2]:
```
print(type(A))
```

```
<class 'numpy.ndarray'>
```

ndarray stands for **N-dimensional array**. This means that this same array type can be used for more than 1D arrays. We'll see how to do that further down.

How can I refer to a **single element** in *A*?

In [3]:
```
print(A[0], A[1], A[2])
```

```
1 2 3
```

Like most data types in python, the indexing in numpy arrays starts from 0.

We said before that an array must contain elements of the same type.
What is the **data type of the elements** in *A*?

In [4]:
```
print(type(A[0]))
```

```
<class 'numpy.int32'>
```

They're not regular integers, but a custom numpy 64-bit integer data type. We could also see this from a built in array variable.

In [5]:
```
print(A.dtype)
```

```
int32
```

Numpy supports a lot of different **data types**. Integers and Unsigned Integers (8, 16, 32, 64 bit), float (half - 16, single - 32 and double precision - 64), complex numbers (16, 32 and 64 bit) and others.

Now that we got that out of the way, how can we **change an element** in *A*?

In [6]:
```
A[0] = 2.9; A[1] = 5; A[2] = 7.1
print(A)
```

```
[2 5 7]
```

Slicing works like lists too.

In [7]:
```
print(A[:2])  # print the first two elements
print(A[-2:])  # print the last two elements
```

```
[2 5]
[5 7]
```

2D Arrays

Two dimensional arrays are essencially matrices, they are the most common form of arrays we will use. We usually refer to the first dimension as *rows* and the second as *columns*. Dimensions are refered to as **axes** in numpy.

Let's create the array:

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

In [8]:
```
B = np.array([[1, 2, 3], [4, 5, 6]])  # we can initialize arrays with known value
print(B)
```

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

The initialization is done through a list for the rows, containing lists of values for the columns.

Now say we want to retrieve the bottom right element (i.e 6). The second row's index is 1 while the third column's index is 2.

In [9]:
```
print(B[1,2])
```

```
6
```

We just separate the two indices with a comma.

Slicing works on each dimension separately.

In [10]:
```
print(B[-1, ::2])  # print the last element from the odd columns
```

```
[4 6]
```

Built-in Functions

There are also a lot of built-in functions in numpy arrays. The most important ones are the following

Array Information:

- `A.dtype` returns the type of the elements in *A*.
- `A.shape` returns the shape of *A*, i.e a tuple containing the size of it's dimensions.
- `A.ndim` returns the rank of *A*, i.e the number of it's dimensions.
- `A.size` returns the number of elements in *A*, i.e the product of the values in `A.shape` .
- `A.nbytes` returns the total bytes consumed by the elements of the array, i.e (bytes in `A.dtype`) `* A.size / 8` .

Array Conversions:

- `A.tolist()` returns the array as a (possibly nested) list.
- `A.tofile()` writes *A* to a file as text or binary (default).
- `A.dump(file)` dumps a pickle of *A* to the specified *file*.
- `A.astype(dtype)` returns the copy of the array, cast to a specified type.
- `A.copy()` returns a copy of the *A*.

Shape Manipulation:

- `A.traspose` or `A.T` returns transposed *A*.
- `A.reshape(shape)` changes the shape of *A* to *shape*.
- `A.swapaxes(axis1, axis2)` returns a view of the array with axis1 and axis2 interchanged.
- `A.flatten()` returns a copy of the array collapsed into one dimension.
- `A.repeat(n)` repeats elements of an array *n* times. e.g. $A = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ for $n = 2$ becomes $A = \begin{pmatrix} 1 & 1 & 2 & 2 & 3 & 3 \end{pmatrix}$

Item Manipulation:

- `A.sort()` sorts *A*, in-place.
- `A.argsort()` returns the indices that would sort *A*.
- `A.nonzero()` returns the indices of the elements that are non-zero.

Calculations:

- `A.max()` returns the largest element in *A*.
- `A.min()` returns the smallest element in *A*.
- `A.sum()` returns the sum of the elements in *A*.
- `A.mean()` returns the mean of the elements in *A*.
- `A.var()` returns the variation of the elements in *A*.
- `A.std()` returns the standard deviation of the elements in *A*.
- `A.prod()` returns the product of the elements in *A*.
- `A.all()` returns True if all elements in *A* are True.
- `A.any()` returns True if any of the elements in *A* are True.

In [11]:
```
print('There are', B.size, 'elements in B.')
print('The shape of B is', B.shape)
print('The rank of B is', B.ndim)
print('\n')

print('Original Array:\n', B)
print('\n')
print('List equivalent:\n', B.tolist())
print('\n')

print('Transposed:\n', B.T)
print('\n')

print('max =', B.max())
print('min =', B.min())
print('sum =', B.sum())
print('mean =', B.mean())
print('sum / size = ', B.sum() / float(B.size))
```

```
There are 6 elements in B.
The shape of B is (2, 3)
The rank of B is 2


Original Array:
 [[1 2 3]
 [4 5 6]]


List equivalent:
 [[1, 2, 3], [4, 5, 6]]


Transposed:
 [[1 4]
 [2 5]
 [3 6]]


max = 6
min = 1
sum = 21
mean = 3.5
sum / size =  3.5
```

Array Creation

There are a lot of ways to create arrays. The basic way we saw up till now (which creates and populates the array) servers only for small arrays. What we tend to do in practice, is to create an array, either empty or containing placeholder values. Then we populate that array through iteration.

In [12]:
```
shp = (256, 256, 3)  # The shape we want our array to have
E = np.zeros(shp, dtype='f')  # array containing zeros
print(E[0,:3,:])
print('\n')
E = np.arange(np.prod(shp), dtype='f').reshape(shp)  # array with values from ran
# The above line consists of three commands:
# the first one, np.prod(shp), returns the product of the values in shp, i.e 3*2*
# the second one, np.arrange(18), creates a 1D array contianing elements from ran
# the last one, E.reshape(shp), changes E's shape to shp.
print(E[0,:3,:])
```

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


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

Array operations

By default most operations performed on numpy arrays are elementwise.

$$Arr1 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, Arr2 = \begin{pmatrix} 11 & 12 & 13 \\ 14 & 15 & 16 \\ 17 & 18 & 19 \end{pmatrix}$$

For example in numpy the product of these two arrays is:

$$Arr1 \cdot Arr2 = \begin{pmatrix} 11 & 24 & 39 \\ 56 & 75 & 96 \\ 119 & 144 & 171 \end{pmatrix}$$

Elementwise operations

In [13]:
```
print('A =                ', A)
print('A + 1 =            ', A + 1)
print('A ** 2 =           ', A ** 2)
O = np.arange((3))
print('O =                ', O)
print('A + O =            ', A + O)
print('A * O =            ', A * O)
print('2 ** (O + 1) - A =', 2 ** (O + 1) - A)
```

```
A =                [2 5 7]
A + 1 =            [3 6 8]
A ** 2 =           [ 4 25 49]
O =                [0 1 2]
A + O =            [2 6 9]
A * O =            [ 0  5 14]
2 ** (O + 1) - A = [ 0 -1  1]
```

These are also equivalent to.

```
np.add(A, O)
np.subtract(A, O)
np.multiply(A, O)
np.divide(A, O)
np.exp(A)
np.sqrt(A)
```

Note that `A * O` is **not** matrix multiplication! It is an **elementwise multiplication**.

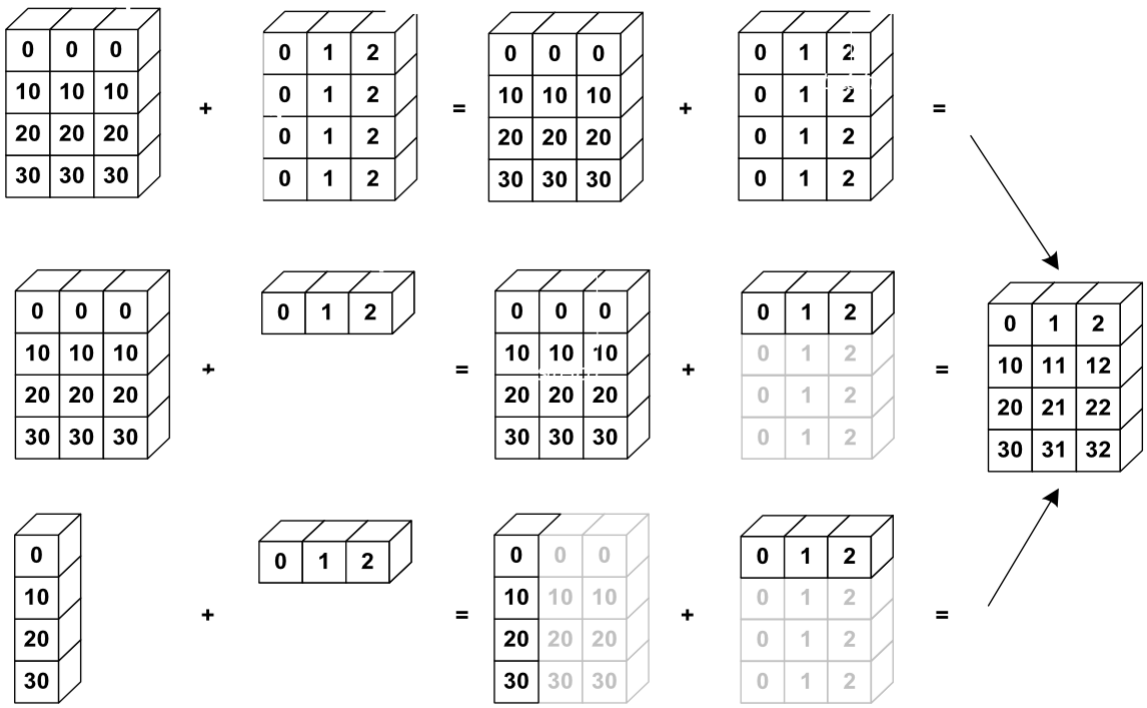
Vecotr/Matrix operations

In [14]:
```
print('matrix mul: A * O = ', np.dot(A, O))
print('same as: ', A.dot(O))
print('outer product: A x O\n', np.outer(A,O))
```

```
matrix mul: A * O =  19
same as:  19
outer product: A x O
 [[ 0  2  4]
 [ 0  5 10]
 [ 0  7 14]]
```

Broadcasting

Some elementwise operations are not defined mathematically. For this reason numpy uses a technique called broadcasting. What this does is it repeats the element that has less axes to match the other. This is illustrated in the image below.



In [15]:
```
A = np.array([[0], [10], [20], [30]]).repeat(3, axis=1)
# Creates a column array with 4 values and repeats 3 times it along the horizonta
B = np.array([[0, 1, 2],]).repeat(4, axis=0)
# Creates an array with 3 values and repeats 4 times it along the vertical axis.
print('{} \n + \n {} \n = \n {}'.format(A, B, A+B))
```

```
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
 +
 [[0 1 2]
 [0 1 2]
 [0 1 2]
 [0 1 2]]
 =
 [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

In [16]:
```
B = np.array([0, 1, 2])
print('{} \n + \n {} \n = \n {}'.format(A, B, A+B))
```

```
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
 +
 [0 1 2]
 =
 [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

In [17]:
```
A = np.array([[0], [10], [20], [30]])
print('{} \n + \n {} \n = \n {}'.format(A, B, A+B))
```

```
[[ 0]
 [10]
 [20]
 [30]]
 +
 [0 1 2]
 =
 [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Input/Output operations

The easiest way of saving/loading numpy arrays is through the built-in functions `np.save` and `np.load`. The default format is a binary NumPy `.npy` format using python pickles.

In [18]:
```
np.save('my_array.npy', B)  # store array B into a file named 'my_array.npy'
B_loaded = np.load('my_array.npy')  # loads array 'my_array.npy' into variable B_
print(B_loaded)
print('original data type:', B.dtype, '\nloaded data type:', B_loaded.dtype)
```

```
[0 1 2]
original data type: int32
loaded data type: int32
```

CSV

While binary `.npy` files might be more memory efficient and faster for saving/loading, they have two major drawbacks. They can't be read by other programms (matlab, R, etc.) and they aren't **human readable**. The most common way of storing data in arrays in the so called CSV format.

CSV stands for *comma-separated values*. This means we store the array in a text file. The values in the columns are separated with *commas*, while rows are separated with *new lines*.

For example:

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Is stored as:

```
1,2,3
4,5,6
```

To store arrays as csv files we can use `np.savetxt` and `np.loadtxt` .

In [19]:
```
np.savetxt('my_array.csv', B, delimiter=',')  # default delimiter is a single spa
B_loaded = np.loadtxt('my_array.csv', delimiter=',')  # since we didn't use de de
print(B_loaded)
print('original data type:', B.dtype, '\nloaded data type:', B_loaded.dtype)
```

```
[0. 1. 2.]
original data type: int32
loaded data type: float64
```

Note that CSVs dont store any extra information about the array other than it's values. This is why when we stored and loaded the array the data type changed to the numpy's default `float64` . On the other hand, when saving as a binary file, this extra information is preserved.

Tips: Automatic reshaping

In [20]:
```
A = np.arange(30)
A.shape = 2, -1, 3 # -1 means 'whatever is needed'
print(A)
```

```
[[[ 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]]]
```

Random

NumPy has excellent support for random numbers.

Regular functions:

- np.random.rand(shp) returns random values in a given shape.
- np.random.random() returns random floats in [0,1).
- np.random.shuffle(x) shuffles the contents of x.

Distributions:

- np.random.binomial(n,p)
- np.random.gamma(shp)
- np.random.geometric(p)
- np.random.multinomial(n,p)
- np.random.poisson()