# Introduction to NumPy

The learning objectives of this section are:

- Understand advantages of vectorized code using Numpy (over standard python ways)
- Create NumPy arrays
  - Convert lists and tuples to numpy arrays
  - Create (initialise) arrays
- Inspect the structure and content of arrays
- Subset, slice, index and iterate through arrays
- Compare computation times in NumPy and standard Python lists

## NumPy Basics

NumPy is a library written for scientific computing and data analysis. It stands for numerical python.

The most basic object in NumPy is the `ndarray` , or simply an `array` , which is an **n-dimensional, homogenous** array. By homogenous, we mean that all the elements in a numpy array have to be of the **same data type**, which is commonly numeric (float or integer).

Let's see some examples of arrays.

In [2]:
```
# Import the numpy library
# np is simply an alias, you may use any other alias, though np is quite standard
import numpy as np
```

In [3]:
```
# Creating a 1-D array using a list
# np.array() takes in a list or a tuple as argument, and converts into an array
array_1d = np.array([2, 4, 5, 6, 7, 9])
print(array_1d)
print(type(array_1d))
```

```
[2 4 5 6 7 9]
<class 'numpy.ndarray'>
```

In [4]:
```
# Creating a 2-D array using two lists
array_2d = np.array([[2, 3, 4], [5, 8, 7]])
print(array_2d)
```

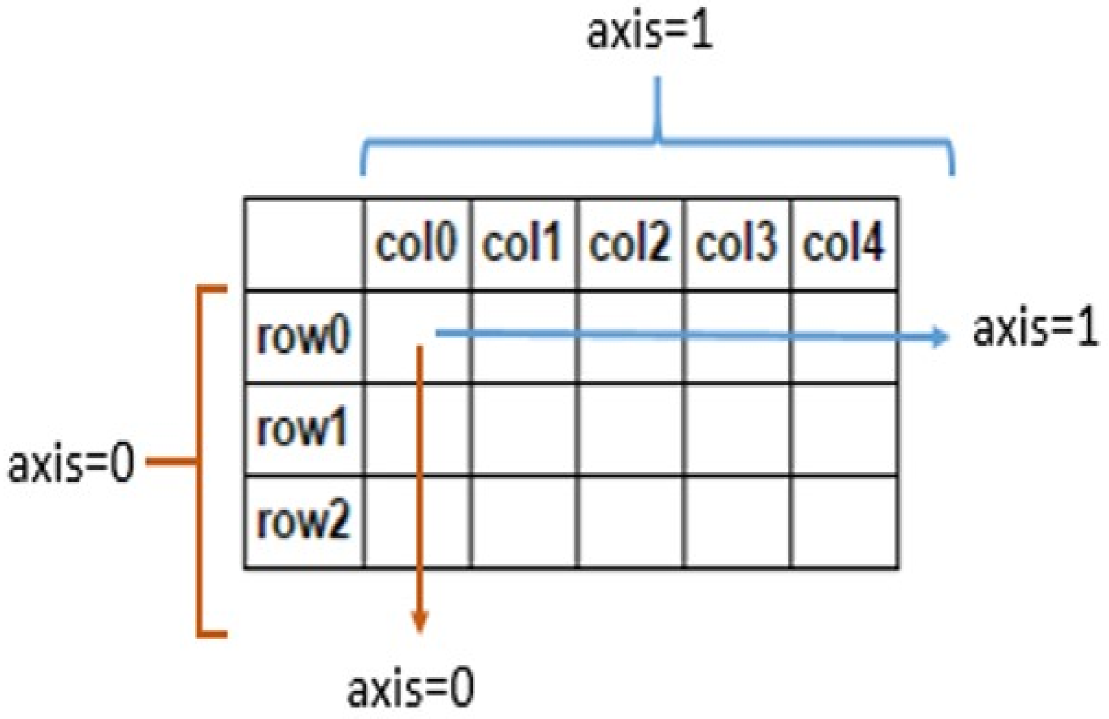
```
[[2 3 4]
 [5 8 7]]
```

In NumPy, dimensions are called **axes**. In the 2-d array above, there are two axes, having two and three elements respectively.

In Numpy terminology, for 2-D arrays:

- `axis = 0` refers to the rows

- `axis = 1` refers to the columns



## Advantages of NumPy

What is the use of arrays over lists, specifically for data analysis? Putting crudely, it is **convenience and speed** :

1. You can write **vectorised** code on numpy arrays, not on lists, which is **convenient to read and write, and concise**.
2. Numpy is **much faster** than the standard python ways to do computations.

Vectorised code typically does not contain explicit looping and indexing etc. (all of this happens behind the scenes, in precompiled C-code), and thus it is much more concise.

Let's see an example of convenience, we'll see one later for speed.

Say you have two lists of numbers, and want to calculate the element-wise product. The standard python list way would need you to map a lambda function (or worse - write a `for` loop), whereas with NumPy, you simply multiply the arrays.

In [5]:
```
list_1 = [3, 6, 7, 5]
list_2 = [4, 5, 1, 7]

# the list way to do it: map a function to the two lists
product_list = list(map(lambda x, y: x*y, list_1, list_2))
print(product_list)
```

```
[12, 30, 7, 35]
```

In [6]:
```
# The numpy array way to do it: simply multiply the two arrays
array_1 = np.array(list_1)
array_2 = np.array(list_2)

array_3 = array_1*array_2
print(array_3)
print(type(array_3))
```

```
[12 30  7 35]
<class 'numpy.ndarray'>
```

As you can see, the numpy way is clearly more concise.

Even simple mathematical operations on lists require for loops, unlike with arrays. For example, to calculate the square of every number in a list:

In [7]:
```
# Square a list
list_squared = [i**2 for i in list_1]

# Square a numpy array
array_squared = array_1**2

print(list_squared)
print(array_squared)
```

```
[9, 36, 49, 25]
[ 9 36 49 25]
```

This was with 1-D arrays. You'll often work with 2-D arrays (matrices), where the difference would be even greater. With lists, you'll have to store matrices as lists of lists and loop through them. With NumPy, you simply multiply the matrices.

## Creating NumPy Arrays

There are multiple ways to create numpy arrays, the most commmon ones being:

- Convert lists or tuples to arrays using `np.array()` , as done above
- Initialise arrays of fixed size (when the size is known)

```
In [8]: # Convert lists or tuples to arrays using np.array()
        # Note that np.array(2, 5, 6, 7) will throw an error - you need to pass a list or
        array_from_list = np.array([2, 5, 6, 7])
        array_from_tuple = np.array((4, 5, 8, 9))

        print(array_from_list)
        print(array_from_tuple)
```

```
[2 5 6 7]
[4 5 8 9]
```

The other common way is to initialise arrays. You do this when you know the size of the array beforehand.

The following ways are commonly used:

- `np.ones()` : Create array of 1s
- `np.zeros()` : Create array of 0s
- `np.random.random()` : Create array of random numbers
- `np.arange()` : Create array with increments of a fixed step size
- `np.linspace()` : Create array of fixed length

In [9]:
```
# Tip: Use help to see the syntax when required
help(np.ones)
```

```
Help on function ones in module numpy.core.numeric:

ones(shape, dtype=None, order='C')
    Return a new array of given shape and type, filled with ones.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    dtype : data-type, optional
        The desired data-type for the array, e.g., `numpy.int8`.  Default is
        `numpy.float64`.
    order : {'C', 'F'}, optional
        Whether to store multidimensional data in C- or Fortran-contiguous
        (row- or column-wise) order in memory.

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and order.

    See Also
    --------
    zeros, ones_like

    Examples
    --------
    >>> np.ones(5)
    array([ 1.,  1.,  1.,  1.,  1.])

    >>> np.ones((5,), dtype=np.int)
    array([1, 1, 1, 1, 1])

    >>> np.ones((2, 1))
    array([[ 1.],
           [ 1.]])

    >>> s = (2,2)
    >>> np.ones(s)
    array([[ 1.,  1.],
           [ 1.,  1.]])
```

In [10]:
```
# Creating a 5 x 3 array of ones
np.ones((5, 3))
```

Out[10]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

In [11]:
```
# Notice that, by default, numpy creates data type = float64
# Can provide dtype explicitly using dtype
np.ones((5, 3), dtype = np.int)
```

Out[11]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

In [12]:
```
# Creating array of zeros
np.zeros(4, dtype = np.int)
```

Out[12]:
```
array([0, 0, 0, 0])
```

In [13]:
```
# Array of random numbers
np.random.random([3, 4])
```

Out[13]:
```
array([[ 0.5383605 ,  0.57178109,  0.69616489,  0.36651827],
       [ 0.49978365,  0.35176881,  0.25195799,  0.56667081],
       [ 0.62282179,  0.5865572 ,  0.18125427,  0.26725554]])
```

In [14]:
```
# np.arange()
# np.arange() is the numpy equivalent of range()
# Notice that 10 is included, 100 is not, as in standard python lists

# From 10 to 100 with a step of 5
numbers = np.arange(10, 100, 5)
print(numbers)
```

```
[10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]
```

In [15]:
```
# np.linspace()
# Sometimes, you know the length of the array, not the step size

# Array of length 25 between 15 and 18
np.linspace(15, 18, 25)
```

Out[15]:
```
array([ 15.   ,  15.125,  15.25 ,  15.375,  15.5  ,  15.625,  15.75 ,
        15.875,  16.   ,  16.125,  16.25 ,  16.375,  16.5  ,  16.625,
        16.75 ,  16.875,  17.   ,  17.125,  17.25 ,  17.375,  17.5  ,
        17.625,  17.75 ,  17.875,  18.   ])
```

## Inspect the Structure and Content of Arrays

It is helpful to inspect the structure of numpy arrays, especially while working with large arrays. Some attributes of numpy arrays are:

- `shape` : Shape of array (n x m)
- `dtype` : data type (int, float etc.)
- `ndim` : Number of dimensions (or axes)
- `itemsize` : Memory used by each array elememnt in bytes

Let's say you are working with a moderately large array of size 1000 x 300. First, you would want to wrap your head around the basic shape and size of the array.

In [16]:
```
# Initialising a random 1000 x 300 array
rand_array = np.random.random((1000, 300))

# Print the first row
print(rand_array[1, ])
```

```
[ 0.76732994  0.28516507  0.10345333  0.84452426  0.38027697  0.37943914
  0.35971136  0.55089254  0.61079053  0.64107306  0.64674768  0.44888241
  0.86850205  0.51426683  0.58543826  0.15111718  0.06274135  0.46907591
  0.63630879  0.48122794  0.19187493  0.7030287   0.89290758  0.91055063
  0.89477124  0.70279072  0.44074611  0.46186643  0.54159695  0.6440544
  0.3957165   0.65119314  0.48610534  0.65977458  0.68551806  0.21220541
  0.29707786  0.62519864  0.17647769  0.98314147  0.59093206  0.66151632
  0.58895999  0.88590139  0.8685559   0.9883252   0.89459641  0.50366793
  0.07559078  0.75926197  0.36447772  0.8533688   0.09735159  0.49114069
  0.18021525  0.84715546  0.38366286  0.02622313  0.74605711  0.13586369
  0.31799587  0.44633302  0.65412178  0.99627805  0.41757153  0.08570335
  0.31578157  0.91397785  0.02555325  0.70910641  0.11160907  0.70250319
  0.49197229  0.55829694  0.86518521  0.43944082  0.05583068  0.74823841
  0.79388085  0.23817489  0.10086487  0.72830127  0.28358495  0.4135227
  0.83385021  0.6459051   0.57532394  0.21340132  0.36659497  0.7416929
  0.2809288   0.47647631  0.42909367  0.81777435  0.51932957  0.65339856
  0.12834393  0.47314972  0.17193208  0.70788945  0.14943493  0.97052028
  0.57043771  0.52011903  0.99610835  0.83203306  0.73723053  0.78665718
  0.9775182   0.743881    0.60279358  0.73756938  0.5177046   0.97314023
  0.17213864  0.00272121  0.09367538  0.97835864  0.34947905  0.45152338
  0.87367407  0.88919074  0.59834799  0.61399075  0.08875363  0.77307253
  0.31021158  0.31834989  0.0261817   0.50584846  0.33664339  0.99380948
  0.21041387  0.76859745  0.39369317  0.94019728  0.80388208  0.26150104
  0.40291998  0.70363167  0.87732735  0.75169225  0.79096407  0.26886768
  0.13300086  0.33553416  0.39345179  0.32347604  0.29581868  0.90525391
  0.82194376  0.53488478  0.49106365  0.30331674  0.87102604  0.26256678
  0.83834867  0.62452721  0.53774213  0.44091098  0.27132409  0.36254457
  0.55213023  0.93613166  0.2218475   0.47249341  0.70868109  0.29415748
  0.89348397  0.65096138  0.94203247  0.65254147  0.52674494  0.69110091
  0.99360579  0.54517031  0.1593096   0.66351029  0.15447795  0.79835919
  0.21387552  0.89971853  0.5955027   0.53612126  0.8418045   0.87872825
  0.35922193  0.38236779  0.92972755  0.01320955  0.15189158  0.67807428
  0.83079828  0.13882951  0.8023863   0.03033067  0.26684051  0.05027809
  0.26698002  0.10233705  0.85907126  0.99148989  0.61603323  0.66855783
  0.86191884  0.00245704  0.42954155  0.43474678  0.95438291  0.23413069
  0.98529592  0.08626932  0.17199833  0.53780941  0.48654334  0.68370418
  0.48525737  0.2643267   0.21436716  0.54432235  0.32319164  0.29233054
  0.01441574  0.94189712  0.0529483   0.11405488  0.45083008  0.75165856
  0.71703894  0.23249926  0.45198615  0.8806958   0.872822    0.62798496
  0.68245892  0.20215751  0.9290579   0.37793263  0.19156749  0.73658761
  0.64874437  0.05422701  0.34859108  0.50367884  0.13074272  0.72789542
  0.93840817  0.02957842  0.13904401  0.67495721  0.21645564  0.95466277
  0.56408525  0.30533271  0.24364057  0.7181483   0.97914201  0.47649146
  0.5204488   0.51149014  0.67652021  0.87946819  0.97425932  0.03573751
  0.24266589  0.20769115  0.59654341  0.44625116  0.06618744  0.39504064
  0.22100789  0.43497512  0.65946797  0.00732933  0.63928075  0.06521243
  0.30790053  0.18320093  0.9550601   0.79368589  0.45540433  0.18434004
  0.10033613  0.40446562  0.44519118  0.79593634  0.04512614  0.70556428
  0.25187817  0.17362854  0.10650895  0.59162577  0.22177357  0.77923098
  0.46002686  0.40879426  0.54929542  0.71532414  0.67052636  0.57138113]
```

In [17]:
```
# Inspecting shape, dtype, ndim and itemsize
print("Shape: {}".format(rand_array.shape))
print("dtype: {}".format(rand_array.dtype))
print("Dimensions: {}".format(rand_array.ndim))
print("Item size: {}".format(rand_array.itemsize))
```

```
Shape: (1000, 300)
dtype: float64
Dimensions: 2
Item size: 8
```

Reading 3-D arrays is not very obvious, because we can only print maximum two dimensions on paper, and thus they are printed according to a specific convention. Printing higher dimensional arrays follows the following conventions:

- The last axis is printed from left to right
- The second-to-last axis is printed from top to bottom
- The other axes are also printed top-to-bottom, with each slice separated by another using an empty line

Let's see some examples.

In [18]:
```
# Creating a 3-D array
# reshape() simply reshapes a 1-D array
array_3d = np.arange(24).reshape(2, 3, 4)
print(array_3d)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

- The last axis has 4 elements, and is printed from left to right.
- The second last has 3, and is printed top to bottom
- The other axis has 2, and is printed in the two separated blocks

## Subset, Slice, Index and Iterate through Arrays

For **one-dimensional arrays**, indexing, slicing etc. is **similar to python lists** - indexing starts at 0.

In [19]:
```
# Indexing and slicing one dimensional arrays
array_1d = np.arange(10)
print(array_1d)
```

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

In [20]:
```
# Third element
print(array_1d[2])

# Specific elements
# Notice that array[2, 5, 6] will throw an error, you need to provide the indices
print(array_1d[[2, 5, 6]])

# Slice third element onwards
print(array_1d[2:])

# Slice first three elements
print(array_1d[:3])

# Slice third to seventh elements
print(array_1d[2:7])

# Subset starting 0 at increment of 2
print(array_1d[0::2])
```

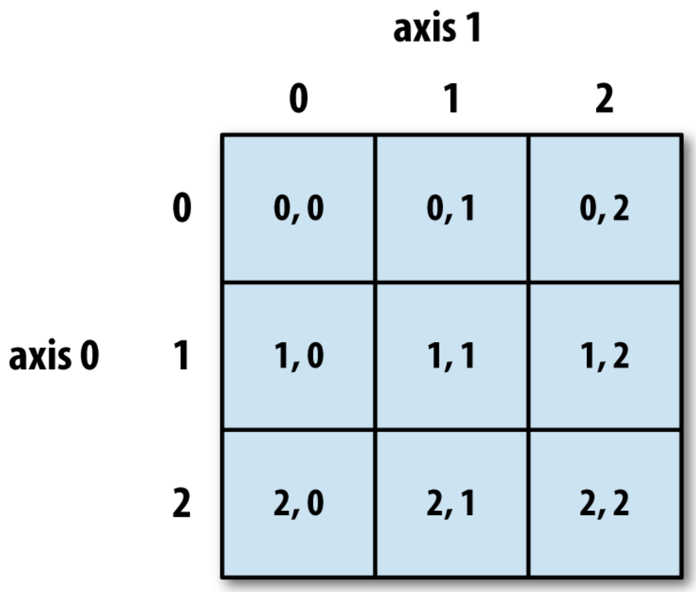
```
2
[2 5 6]
[2 3 4 5 6 7 8 9]
[0 1 2]
[2 3 4 5 6]
[0 2 4 6 8]
```

In [21]:
```
# Iterations are also similar to lists
for i in array_1d:
    print(i**2)
```

```
0
1
4
9
16
25
36
49
64
81
```

**Multidimensional arrays** are indexed using as many indices as the number of dimensions or axes. For instance, to index a 2-D array, you need two indices - `array[x, y]` .

Each axes has an index starting at 0. The following figure shows the axes and their indices for a 2-D array.



In [22]:
```
# Creating a 2-D array
array_2d = np.array([[2, 5, 7, 5], [4, 6, 8, 10], [10, 12, 15, 19]])
print(array_2d)
```

```
[[ 2  5  7  5]
 [ 4  6  8 10]
 [10 12 15 19]]
```

In [23]:
```
# Third row second column
print(array_2d[2, 1])
```

```
12
```

In [24]:
```
# Slicing the second row, and all columns
# Notice that the resultant is itself a 1-D array
print(array_2d[1, :])
print(type(array_2d[1, :]))
```

```
[ 4  6  8 10]
<class 'numpy.ndarray'>
```

In [25]:
```
# Slicing all rows and the third column
print(array_2d[:, 2])
```

```
[ 7  8 15]
```

In [26]:
```
# Slicing all rows and the first three columns
print(array_2d[:, :3])
```

```
[[ 2  5  7]
 [ 4  6  8]
 [10 12 15]]
```

**Iterating on 2-D arrays** is done with respect to the first axis (which is row, the second axis is column).

In [27]:
```
# Iterating over 2-D arrays
for row in array_2d:
    print(row)
```

```
[2 5 7 5]
[ 4  6  8 10]
[10 12 15 19]
```

In [28]:
```
# Iterating over 3-D arrays: Done with respect to the first axis
array_3d = np.arange(24).reshape(2, 3, 4)
print(array_3d)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

In [29]:
```
# Prints the two blocks
for row in array_3d:
    print(row)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

## Compare Computation Times in NumPy and Standard Python Lists

We mentioned that the key advantages of numpy are convenience and speed of computation.

You'll often work with extremely large datasets, and thus it is important point for you to understand how much computation time (and memory) you can save using numpy, compared to standard python lists.

Let's compare the computation times of arrays and lists for a simple task of calculating the element-wise product of numbers.

In [30]:

```
## Comparing time taken for computation
list_1 = [i for i in range(1000000)]
list_2 = [j**2 for j in range(1000000)]

# list multiplication
import time

# store start time, time after computation, and take the difference
t0 = time.time()
product_list = list(map(lambda x, y: x*y, list_1, list_2))
t1 = time.time()
list_time = t1 - t0
print(t1-t0)


# numpy array
array_1 = np.array(list_1)
array_2 = np.array(list_2)

t0 = time.time()
array_3 = array_1*array_2
t1 = time.time()
numpy_time = t1 - t0

print(t1-t0)

print("The ratio of time taken is {}".format(list_time/numpy_time))
```

```
0.4139885902404785
0.015633344650268555
The ratio of time taken is 26.481127327629594
```

In this case, numpy is **an order of magnitude faster** than lists. This is with arrays of size in millions, but you may work on much larger arrays of sizes in order of billions. Then, the difference is even larger.

Some reasons for such difference in speed are:

- NumPy is written in C, which is basically being executed behind the scenes
- NumPy arrays are more compact than lists, i.e. they take much lesser storage space than lists

The following discussions demonstrate the differences in speeds of NumPy and standard python:

1. https://stackoverflow.com/questions/8385602/why-are-numpy-arrays-so-fast (https://stackoverflow.com/questions/8385602/why-are-numpy-arrays-so-fast)
2. https://stackoverflow.com/questions/993984/why-numpy-instead-of-python-lists (https://stackoverflow.com/questions/993984/why-numpy-instead-of-python-lists)