Numpy is a Python 'library' designed, at its core, as the name would suggest, for performing numerical computations in a vectorized manner. We will explore some of these features throughout this course, but this specific lecture focuses on the more flexible indexing options offered by **numpy** over baseline Python.

As usual, for working with **numpy**, we will need the standard import syntax, c where the *as np* part of the command provides a local (inside the *.py* file) 'nickname' for the **numpy** package.

The fundamental object class in **numpy** is the **array** class. Although the declaration and construction of an *array* object is very similar to that of a list in baseline Python, an *array* could be intuitively described as more structured. *Arrays* are the equivalent of vectors, matrices and tensors (higher dimensional matrices) in Python.

The following code

```
x = np.array([[ -5,    2,   0,  -7],
              [ -1,    9,   3,   8],
              [ -3,   -3,   4,   6]])
```

is a typical example of initializing an *array* x via the constructor method *np.array()*. In this specific case, x will be a matrix with 3 rows and 4 columns. Note that in terms of syntax, the object passed to the method *np.array()* is essentially a list of lists (note the nested square brackets in the declaration)), where the inner lists are the rows of the desired matrix and the outer list is just a list of the 'rows'.

As usual, we can call

```
print(x)
```

to inspect the matrix x. It is worth pointing out the visual distinction between printing a *list* and an *array*. Namely, a *list* is printed with commas separating the elements and no ellipsis is done in the printing (i.e., if you are printing a long *list*, all the elements will be displayed). In contrast, the elements of an *array* are separated by blank spaces and if the vector or matrix is too large to display neatly in the print window, ellipsis dots will be used.

Baseline Python comes with what is called 'basic' indexing, i.e., indexing by integers or by slices. One of the benefits of **numpy** is the flexibility of having available 'advanced' indexing alongside 'basic' indexing. The main forms of 'advanced indexing' are indexing by a *list* (or *array*) and logical indexing. Additionally, **numpy** introduces higher dimensional structures (e.g. matrices or tensors) which come with their own indexing particularities.

A simple example of basic indexing into a matrix would be something like

```
print(x[0,1])
```

which would return the value stored in x in row 0 and column 1, i.e., the value 2. Note that the same result is obtained by calling

```
print(x[(0,1)])
```

This is not accidental. In fact, whenever we index into a dimension 2 or higher object, the indices used for indexing are actually interpreted as a tuple, since the ordering of the dimensions matters. It is important to remember that in Python row indices always come before column indices.

Another feature of indexing in Python in general is the fact that indexing tuples are auto-completed with full slices (i.e., Python will pretend the missing components of an indexing tuple are ':'). So

```
print(x[2])
```

will return '[-3 -3 4 6]' (the second row), just as the command 'print(x[2,:])' would.

If we want to use this short hand notation for columns, we can employ the *ellipsis* operator (especially useful in dimensions 3 and higher). For example,

```
print(x[...,1])
```

would return the column of index 1 from x, i.e., '[2 9 -3]'. This also illustrates the fact that the 'native vector' objector in **numpy** is the row vector. A column vector is always interpreted as a list of rows with a single element.

An instance of indexing by a list ('advanced' indexing) would be

```
print(x[[0,1]])
```

This would return the first two rows of x (i.e., rows with indices 0 and 1 from x). Contrast this case, where the indexing set is the list '[0,1]', to the previous case where the indexing was done with the tuple '(0,1)'.

As a general rule of thumb, in cases where we need to index into a list or array and the indices we need to access are regular (e.g. they are in an arithmetic progression or we need to take all the values from some point on or up to a point), 'basic' indexing via slices should suffice. In cases where the index set is irregular, for example if we have a list of consecutive numbers and we need to access only the values that have prime indices, indexing by a list might be the most efficient option. Since 'advanced' indexing is not available for baseline *lists* in Python, this could imply converting to an *array* (by using the constructor method 'np.array()') and converting the resulting *array* back to a *list* (using the list constructor method 'list()').

Arguably the most important (and most common) usage of 'advanced' indexing is logical indexing. Calling

```
print(x>0)
```

will return a logical *array* of the same size of x (the size of x can directly be evaluated with the syntax 'x.shape') which will contain 'True' in the positions where the corresponding value in x was strictly positive and the value 'False' everywhere else.

The utility of this is that the command

```
print(x[x>0])
```

will return a row *array* with only the positive values contained in x.

This can be used to update the values of x that satisfy a logical condition. For example

```
x[x>0] = 0
```

would set all the positive values in x to 0.

One other aspect of *arrays* that is worth knowing is the fact that indexing produces 'images' of the original array instead of new variable.

```
u = x[:,0]
x[0,0] = 5
print(u)
```

With this code, u is the first column of x. But it is pointing to the same memory as x. Indeed,

```
print(np.shares_memory(u, x))
```

will return the logical value 'True'.

After we update the first entry in the first column of x to 5 (as opposed to the original value -5), this update will be carried over to u as well, even though u was 'constructed' before the change to x. Printing u will return the *array* '[5 -1 -3]'.

This behavior can be avoided by the using the 'copy()' method. To see this, replace the line 'u = x[:, 0]' from above with the command 'u = x[:, 0].copy()'. A rather amusing observation is that the forcing of an instantiation of a different variable (with different memory allocation) can also be achieved (in the numerical entry case) by the command 'u = x[:, 0] + 0'. However, incrementation does not allocate new memory.

```
u = x[:,0]
print(np.shares_memory(u, x))
u += 0
print(np.shares_memory(u, x))
u = u + 0
print(np.shares_memory(u, x))
```

In this case, the first two print-outs will be 'True', but the last will be 'False'.