# Lab week 3

In week 2 we introduced lists and looked at `for` loops.

## Script

Last time we saw lists. A list contains many objects. Likewise, there are many mathematical things that contain many objects.

Sets are one type. If you add a set to another set you get a new set containing the objects in both sets. The new set is larger (generically) than either individual set.

Vectors are another type. If you add one vector to another vector you get a new vector whose components are the sum of the individual vector components. The new vector has the same size (in terms of number of components) as the individual vectors.

How does a list act?

```
list_1 = [1, 2]
list_2 = [3, 4]
print(list_1 + list_2)
```

```
[1, 2, 3, 4]
```

A list acts like the description of a set, not like a vector.

Consider what multiplying a container by a scalar number should do for a set and a vector. Then check a list:

```
print(2 * list_1)
```

```
[1, 2, 1, 2]
```

Again, lists act more like sets than vectors.

This is fine when we want to work with general sets. However, we will also frequently need (as in Linear Algebra) vectors, and matrices, and so on. In Python, we can use `numpy` for this: in particular we can use its `array` object. These can be created directly from lists.

```
import numpy
vector_1 = numpy.array(list_1)
vector_2 = numpy.array(list_2)
print(vector_1)
print(vector_1 + vector_2)
print(2 * vector_1)
```

```
[1 2]
[4 6]
[2 4]
```

We see that an `array` is displayed much like a list (using spaces rather than commas to separate entries). However, it acts as a vector would: addition of a pair of two-dimensional vectors produces another two-dimensional vector, as does multiplication of a two-dimensional vector by a scalar.

Note that the `len` function and the indexing operations act the same as on lists:

```
print(len(vector_1))
print(vector_1[0])
```

```
2
1
```

```
numbers = [1, 2, 3, 4, 5]
vector_n = numpy.array(numbers)
print(vector_n)
print(vector_n[-1])
print(vector_n[1:4])
```

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

There are some restrictions with `numpy` arrays. They are meant to represent vectors, matrices, and similar higher dimensional objects. So if you nest lists within lists you must do so consistently:

```
matrix_wrong_1 = numpy.array([1, 2, [3, 4]])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call
last)

<ipython-input-6-ce621beadb7d> in <module>()
----> 1 matrix_wrong_1 = numpy.array([1, 2, [3, 4]])


ValueError: setting an array element with a sequence.
```

```
matrix_right_1 = numpy.array([[1, 2], [3, 4]])
print(matrix_right_1)
```

```
[[1 2]
 [3 4]]
```

You should also use only a single type of object in a `numpy` `array`: do not mix numbers with strings, for example.

With higher dimensional objects like matrices you can access them in two ways. Either use the "list of lists" approach:

```
print(matrix_right_1[0][0])
```

```
1
```

The alternative, which is preferred, is to treat it as a single object that takes multiple indexes:

```
print(matrix_right_1[0, 0])
```

```
1
```

Slicing works with higher dimensional objects:

```
print(matrix_right_1[:, 0])
print(matrix_right_1[1, :])
```

```
[1 3]
[3 4]
```

There are a number of utility functions for creating matrices:

```
print(numpy.ones(3))
print(numpy.ones([2, 2]))
print(numpy.zeros([2, 2]))
print(numpy.identity(3))
print(numpy.random.rand(2, 2))
```

```
[ 1.  1.  1.]
[[ 1.  1.]
 [ 1.  1.]]
[[ 0.  0.]
 [ 0.  0.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
[[ 0.11645041  0.82370944]
 [ 0.69588563  0.87133502]]
```

Note that the conventions of the functions are not always consistent.

Exercise

In the Gaussian Elimination algorithm a matrix (or augmented matrix) is reduced to triangular form by subtracting rows from each other. Write a function `row_elimination` that takes a matrix `A` and two row numbers `r1` and `r2`. It should return the matrix `A` with $\epsilon\times$ `r1` subtracted from row `r2`. Here `epsilon` should make the matrix entry $a_{r_2,r_1} \to 0$: that is,

$$\epsilon = \frac{a_{r_2,r_1}}{a_{r_1,r_1}}.$$

Check on the matrix

$$A = \begin{pmatrix} 1 & 4 & 3 \\ 2 & -7 & 5 \\ 1 & 2 & 6 \end{pmatrix}.$$

Can you use or extend your function to write a function that takes `A` and returns the triangular form?

The `numpy` package also contains a lot of useful Linear Algebra functions in its `linalg` subpackage. For example, matrix multiplication and dot products:

```
A = numpy.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
B = numpy.array([[7, 3, 1], [6, 1, 2], [3, 5, 2]])
d = numpy.array([5, 3, 1])

print(numpy.dot(A, B))
print(numpy.dot(A, d))
print(numpy.dot(d, d))
```

```
[[28 20 11]
 [76 47 26]
 [97 29 23]]
[14 41 59]
35
```

Computing determinants:

```
print(numpy.linalg.det(A))
```

```
27.0
```

Solving linear systems of equations, such as $A\mathbf{x} = \mathbf{d}$:

```
print(numpy.linalg.solve(A, d))
```

```
[ -6.33333333e+00   5.66666667e+00  -1.40980702e-16]
```

Exercise

Define three matrices:

$$A = \begin{pmatrix} 1 & 4 & 3 & 2 \\ 2 & -7 & 5 & 1 \\ 1 & 2 & 6 & 0 \\ 2 & -10 & 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 2 & -4 & -2 \\ -3 & 1 & 5 & 1 \\ 4 & -1 & 1 & 3 \\ 2 & 1 & -1 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} 3 & 0 & 0 & 0 \\ 7 & -6 & 0 & 0 \\ 2 & -1 & -1 & 0 \\ -4 & -2 & 2 & -2 \end{pmatrix}.$$

Compute the ranks of all matrices. Compute the determinants of

$$A^T B^{-1}, \qquad B/4, \qquad C, \qquad 2B + C.$$