# Lab 6 Summary

## Introduction to numpy and matplotlib

- Study *Python Data Science Handbook* Chapter 2: Introduction to Numpy
- Study *Python Data Science Handbook* Chapter 4: Simple Line Plots / Simple Scatterplots

*"If I have seen further, it is by standing on the shoulders of Giants." — Sir Isaac Newton*

Having studied some Python programming skills, we can now start using Data Science libraries in Python.

### 1. Lists vs Numpy arrays

In Python, the List data structure is ordered and mutable. We can use slices to access the elements and we can change the values held by each element of the list.

Lists are the most general data structure in Python. Each element has its own type. This makes it very SLOW to process.

Also, we cannot do basic things like add 1 to each element of a list.

The numpy library was developed to make processing of lists of numbers very FAST.

We import numpy with a shortened name called an alias (by convention this is np).

One simple way to create a numpy array, we can cast it from a list.

```
import numpy as np
print(np.__version__)
N = np.array([2,3,5,7,11])
print(N)
print(type(N))
```

We can add the elements in one numpy array to the elements in another numpy array (of the same length). This is something we cannot directly do with lists.

```
P = np.array([1,2,3,4,5])
print(P)
print(N+P)
```

We often create a numpy array as a range (or sequence) of values. Numpy provides a method to make this much faster.

```
A = np.array(range(1,10,2))
print(A)
B = np.arange(1,10,2)
print(B)
```

There are other ways to build commonly used numpy arrays.

```
C = np.zeros(10)
print(C)
```

## 2. Numpy values and types

A numpy array has type *numpy.ndarray*.

Elements of a numpy array also have a type, e.g., *numpy.int32*, which you will notice is not the same as int or float.

Each element of a numpy array has the SAME type, called its *dtype*.

```
D = np.array([1,2,3])
print(D)
print(D.dtype)
```

The dtype *int32* is a 32-bit integer. It can only take values from -2^31 = -2147483648 to 2^31-1 = 2147483647. This is very fast to work with.

The Python type *int* can take the value of any integer (however large you like). It is very slow to work with large integers.

We can set the dtype of a numpy array when it is created.

```
E = np.array([1,2,3,127], dtype=np.int8)
print(E)
print(E.dtype)
print(E+1)
```

The dtype *int8* is an 8-bit integer. It can only take values from -2^7 = -128 to 2^7-1 = 127.

Notice what happens when we add 1 to 127 (this is called “overflow”).

The rules on how these values behave are governed by an IEEE standard.

## 3. Numpy array attributes

For a list L, we have the length of the list given by len(L).

```
F = np.array([1,2,3])
print(F)
print(F.ndim)   # 1 dimensional
print(F.shape)  # a tuple showing 1 dimension and 3 elements in that dimension
print(F.size)   # total number of elements
print(F.dtype)  # numpy data type of the elements
```

In 7143CEM we will work with one-dimensional numpy arrays. We can think of these as a single row of numbers or other values.

In 7144CEM we will introduced two-dimensional numpy arrays. We can think of these as a grid (or matrix) of numbers or other values, having rows and columns.

## 4. Numpy array slices

When we take a slice from a list we are returned a copy of the sublist.

```
L = [1,2,3,4,5]
print(L)
A = L[0:2]
print(A)
A[0] = 17
print(A)
print(L)
```

We can access numpy arrays using slices in the same way as lists.

*Warning!* – A slice from a numpy array does not return a copy. Instead it returns only a "view".

If we modify the view then we also modify the original.

```
L = np.array([1,2,3,4,5])
print(L)
A = L[0:2]
print(A)
A[0] = 17
print(A)
print(L)
```

If we want a copy, then we need to use the numpy method copy.

```
L = np.array([1,2,3,4,5])
print(L)
A = L[0:2].copy()
print(A)
A[0] = 17
print(A)
print(L)
```

## 5. Numpy vector operations and aggregation

Main idea – loops are very slow so we want to be able to apply an operation element-wise.

```
A = np.array([1,2,3,4])
print(A)
print(A.dtype)
print(A**2)
print(A**2 + 3*A + 9)
B = (A>2)
print(B)
print(B.dtype)
```

Numpy arrays also have methods that apply to summarise an array.

```
C = np.random.randint(1,6,100)
print(C)
print(np.min(C))
print(np.max(C))
print(np.sum(C))
print(np.mean(C))
print(np.median(C))
```

Often we have “missing data”. Rather than record this as a particular number (say 0 or 999), we can use the special value np.nan which is the IEEE representation of “Not a Number”.

```
D = np.array([1,2,np.nan])
print(D)
print(np.isnan(D))
print(np.nanmean(D))
```

## 6. Simple plots using matplotlib

The Python library matplotlib provides a way to build graphical plots.

```
import matplotlib
print(matplotlib.__version__)
```

To do some plotting we need to work with matplotlib.pyplot which is quite a lot to get right when typing so we use an alias (by convention use plt).

```
import matplotlib.pyplot as plt
plt.figure()    # wipes the plotting canvas
# put code here to build a plot
plt.show()
```

*Barchart*

```
names = ['fred','alice']
height = [23,32]
plt.figure()
plt.barh(names,height)
plt.title('Heights of people')
plt.xlabel('Person')
plt.ylabel('Height (cm)')
plt.show()
```

*Line plot*

```
x = np.array([1,2,3])
y = np.array([4,2,9])
```

```
plt.figure()
plt.plot(x,y,'g.')   # g. is green dot
plt.plot(x,y,'r-.')   # r- is red line
plt.ylim(-2,12)
plt.grid()
plt.show()
```

Some comment colours: r=red, g=green, b=blue, k=black, c=cyan, m=magenta, y=yellow

Some common line styles: line -, dashed line --, dash dot line -.,dotted line :

Other basic plots: https://matplotlib.org/stable/tutorials/introductory/sample_plots.html

## Summary

- Lists are slow, numpy arrays are fast.
- A slice of a numpy array is a *view* not a *copy*.
- Operators apply to numpy arrays element-wise (vector operations).
- Aggregation calculates a summary value for a numpy array.
- Matplotlib can be used to produce simple plots (we will look more a plots later).