Episode 7
NumPy

NumPy is a package that introduces an important new datatype called an *n-dimensional array* or *ndarray*. In general, an array is similar to a list, but its elements are of one type and its size is fixed. As for lists, elements of arrays are accessed through their indices, which must be integers.

NumPy

The NumPy ndarray is a true multidimensional array. Each element is individually addressed by a tuple of indices. The number of indices required to specify an element is the *rank* of the array. The elements of the tuple are called *dimensions* or, in NumPy terminology, *axes*. A one-dimensional array is analogous to a mathematical vector, a two-dimensional array has rows and columns like a table or matrix, and so forth. Each dimension has a lower bound (always zero in Python) and an upper bound. The tuple of dimensions is the *shape* of the array, and the total number of elements is the *size* of the array. The rank of the array is unlimited, and the size is limited only by the capacity of the computer's memory.

NumPy provides a number of functions to create arrays. Unlike lists, we do not create empty arrays and expand them; we expect to know the shape when we set up the array.

In all examples, assume
```
import numpy as np
```
is at the top of the file.

We can convert a list to an array:

```
A=np.array([1.,2.,3.,4.,5.,6.,7.])
```

To find the size we can use
```
A.size
```
Which is `7` as we'd expect. The shape is
```
A.shape
```
which returns
```
(7,)
```
This is because the shape is a tuple, and Python represents a tuple with a single element in the above format.

We address elements of this array with indices in square brackets, similar to a list
```
A[0]
A[3]
```

For a two-dimensional array we can start from a two-dimensional list

```
B=np.array([[1.,2.,3.,4.],[5.,6.,7.,8.]])
```

Check the size and shape of this array. Also check len(B). The length of an array is the number of rows, regardless of the other dimensions.

We reference an element of B with two indices:
```
B[1,1]
```
This is the item in the second row and second column.

More usually we use one of the built-in NumPy array constructors to specify the shape and initialize to zeros, ones, nothing in particular, or values in a uniform random distribution:

```
X=np.zeros(100)
Y=np.zeros((10,10))
Z=np.ones((3,4,5))
W=np.empty((100,))
V=np.random((4,4))
```

Notice that in general, the argument to these constructors is a tuple, but in the case of rank-1 arrays either `(100)` or `((100,))` will work. For higher-dimensional arrays it is very easy to try
```
XX=np.zeros(100,100)
```
but that is an error.

Unlike most Python data structures, we can declare NumPy arrays to be of a specific type. If we do not specify then the generic constructors shown above will set the type to (double precision) floating point. If we want a different type we can use an optional argument dtype to declare the type.

M_int=np.zeros((10,10),dtype=int)

In the case of converting from a numerical list to an array, Python will type by inference unless a dtype is present.

```
z=np.array([0,1,2,3])   #integers
zz=np.array([0,1,2,3],dtype=float) #float (64 bits)
M =np.array([True,True,False,True,False]) #Boolean
N=np.zeros((5,),dtype=bool)
```

Print each of the following arrays to see the difference.

```
A=np.zeros((10,))
```

```
IM=np.zeros((10,),dtype='int')
mask=np.zeros((10,),dtype='bool')
```

NumPy provides a function similar to `range`, but whereas `range` returns a list, `arange` returns an array. In addition, `arange` is not restricted to integers but can return any numerical type.

These are similar:

```
v = np.array(range(100))

v = np.arange(100)
```

Like range, `arange` can take an increment, which is 1 if it is not specified. With no other specification of type, it sets the type based on its argument or arguments.

```
w=np.arange(10.)
```

It can take a lower bound

```
x=np.arange(1.,101.)
```

It can also take an increment, in which case the lower bound must be present as well.

```
y=np.arange(0.,1.01,.01)
```

Notice that even for floating-point numbers, the upper bound is not included.

It can also take a dtype argument

```
z=np.arange(10,dtype=float)
```

Arange always generates a one-dimensional array. To create a multidimensional array, use reshape:

```
M=np.arange(1.,26.).reshape(5,5)
```

Arrays can be sliced in the usual way. The upper-bound rule still applies!

```
import numpy as np
 A=np.zeros((100,100))
 B=A[0:11,:]
 S1 = 3
 E1 = 5
 S2 = 25
 E2 = 30
 C = A[S1:E1,S2:E2]
 Col_2 = A[:,1]  #second column
```

Negative indices behave in a way that might seem confusing at first.

```
v=np.array([0,1,2,3,4])
v[-1]
v[:-1] #note element versus slice
```

This is due to the second-bound rule again. The second bound argument is never included in ranges. Since `-1` stands for the last element in the array, in the slice example above it is not included.

```
u=np.arange(25).reshape(5,5)
u[:-1,:] #note what is missing in the matrix
```

NumPy provides many built-in functions to work with ndarrays. In particular, all the arithmetic operators and mathematical functions work *elementwise* on an ndarray. A function that operates elementwise on an array of any rank is called a *ufunc* (universal function).

```
A = np.arange(0, 2*np.pi, np.pi/2)
sin_A= np.sin(A)
```

The math function does not, so
`math.sin(A)`
will produce an error.

Watch out for elementwise division. The division operator is not a linear-algebra solver!

```
A = np.ones(4)
B = np.arange(1, 5)
C = A/B #element-wise division!
```

NumPy can find sums and products, either for the entire array or for a subset of the axes. Note that if an axis is specified, that axis is eliminated; so the result is another array of rank one lower than the original, with a shape consisting of the array minus the selected axis. Try the following:

```
M=np.arange(1.,25.).reshape(4,3,2)
print M.sum()
N=M.sum(0)
print N.shape
print N
K=M.sum(1)
print K.shape
print K
L=M.sum(2)
print L.shape
print L
```

The NumPy package also provides a large number of functions that carry out other operations on arrays. These include functions to transpose, reorganize, split, and join arrays; statistical functions; fast-fourier transform; random sampling with a large selection of probability distributions; and many others. The student should refer to the documentation at http://www.numpy.org for details. Some of the most important of these built-in functions are those which *vectorize* chunks of code; by this we mean that we use array operations instead of loops. In most interpreted languages, including Python, loops are very slow, and eliminating them can speed up the processing time significantly. The `sum` function above is an example; we find the sum by applying the appropriate NumPy built-in rather than writing loops with accumulator variables. Another example is finding the location in an array of the maximum or minimum; we could write a loop that does comparisons, or we could use argmax (or argmin) together with a built-in that returns the array of indices corresponding to some condition being `True`, for example:

```
max_coords=np.where(A==A.max())
```

NumPy is a very powerful tool for numerical computing and a careful study of its capabilities is well worth the time. One of the best tutorial is probably NumPy's own:
https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
The student should peruse the reference at
https://docs.scipy.org/doc/numpy/reference/routines.html
to see what the many built-in functions can do.