# 1 Arrays and Matrices

***NumPy*** provides the most important data types for econometrics, statistics and numerical analysis – **arrays** and **matrices**. The difference between these two data types are:

- Arrays can have 1, 2, 3 or more dimensions, and matrices **always** have 2 dimensions. This means that a 1 by n vector stored as an array has 1 dimension and n elements, while the same vector stored as a matrix has 2-dimensions where the sizes of the dimensions are 1 and n (in either order).
- Standard mathematical operators on arrays operate element-by-element. This is not the case for matrices, where multiplication ($*$) follows the rules of *linear algebra*.
- 2-dimensional arrays can be multiplied using the rules of linear algebra using dot.
- Similarly, the function multiply can be used on two matrices for *element-by-element* multiplication.
- Arrays are more common than matrices, and all functions are thoroughly tested with arrays.
- Functions should also work with matrices, but an occasional strange result may be encountered.
- Arrays can be quickly treated as a matrix using either `asmatrix` or `mat` without copying the underlying data.

The best practice is to use arrays and to use the `asmatrix` view when writing linear algebra-heavy code. It is also important to test any custom function with both arrays and matrices to ensure that false assumptions about the behavior of multiplication have not been made.

## 1.1 Arrays

Arrays are the base data type in NumPy, are are arrays in some ways similar to lists since they both contain collections of elements. The focus of this section is on *homogeneous* arrays containing numeric data – that is, an array where all elements have the same numeric type

Additionally, arrays, unlike lists, are always rectangular so that all rows have the same number of elements.

Arrays are initialized from lists (or tuples) using array. Two-dimensional arrays are initialized using lists of lists (or tuples of tuples, or lists of tuples, etc.), and higher dimensional arrays can be initialized by further nesting lists or tuples.

```
>>> x = [0.0, 1, 2, 3, 4]
>>> y = array(x)
>>> y
array([ 0., 1., 2., 3., 4.])
>>> type(y)
numpy.ndarray
```

Two (or higher) -dimensional arrays are initialized using nested lists.

```
>>> y = array([[0.0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
>>> y
array([[ 0., 1., 2., 3., 4.],
[ 5., 6., 7., 8., 9.]])
>>> shape(y)
(2L, 5L)
>>> y = array([[[1,2],[3,4]],[[5,6],[7,8]]])
>>> y
array([[[1, 2],
[3, 4]],
[[5, 6],
[7, 8]]])
>>> shape(y)
(2L, 2L, 2L)
```

## 1.2 Matrix

Matrices are essentially a subset of arrays, and behave in a virtually identical manner. The two important differences are:

- Matrices always have 2 dimensions
- Matrices follow the rules of linear algebra for Matrix Multiplication ($*$)

## 1.3 1-dimensional Arrays

A vector

$$x = [12345]$$

is entered as a 1-dimensional array using

```
>>> x=array([1.0,2.0,3.0,4.0,5.0])
array([ 1., 2., 3., 4., 5.])
>>> ndim(x)
1
```

If an array with 2-dimensions is required, it is necessary to use a trivial nested list.

```
>>> x=array([[1.0,2.0,3.0,4.0,5.0]])
array([[ 1., 2., 3., 4., 5.]])
>>> ndim(x)
2
```

A matrix is always 2-dimensional and so a nested list is not required to initialize a a row matrix

```
>>> x=matrix([1.0,2.0,3.0,4.0,5.0])
>>> x
matrix([[ 1., 2., 3., 4., 5.]])
>>> ndim(x)
2
```

Notice that the output matrix representation uses nested lists ([[ ]]) to emphasize the 2-dimensional structure of all matrices. The column vector,

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

is entered as a matrix or 2-dimensional array using a set of nested lists

```
>>> x=matrix([[1.0],[2.0],[3.0],[4.0],[5.0]])
>>> x
matrix([[ 1.],
[ 2.],
[ 3.],
[ 4.],
[ 5.]])
>>> x = array([[1.0],[2.0],[3.0],[4.0],[5.0]])
>>> x
array([[ 1.],
[ 2.],
[ 3.],
[ 4.],
[ 5.]])
```

## 1.4 Accessing Elements of an Array

Four methods are available for accessing elements contained within an array: scalar selection, slicing, numerical indexing and logical (or Boolean) indexing. Scalar selection and slicing are the simplest and so are presented first.

## 1.5 2-dimensional Arrays

Matrices and 2-dimensional arrays are rows of columns, and so

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

is input by enter the matrix one row at a time, each in a list, and then encapsulate the row lists in another list.

```
>>> x = array([[1.0,2.0,3.0],[4.0,5.0,6.0],[7.0,8.0,9.0]])
>>> x
array([[ 1., 2., 3.],
[ 4., 5., 6.],
[ 7., 8., 9.]])
```

## 1.6 Multidimensional Arrays

- Higher dimensional arrays are useful when tracking matrix valued processes through time, such as a time varying covariance matrices.
- Multidimensional (N -dimensional) arrays are available for N up to about 30,
- depending on the size of each matrix dimension.
- Manually initializing higher dimension arrays is tedious and error prone, and so it is better to use functions such as `zeros((2, 2, 2))` or `empty((2, 2, 2))`.

## 1.7 Concatenation

Concatenation is the process by which one vector or matrix is appended to another. Arrays and matrices can be concatenation horizontally or vertically

## 1.8 Accessing Elements of an Array

- Four methods are available for accessing elements contained within an array: scalar selection, slicing, numerical indexing and logical (or Boolean) indexing.
- Scalar selection and slicing are the simplest and so are presented first.
- Numerical indexing and logical indexing both depends on specialized functions and so these methods are discussed in Chapter 12.

## 1.9 The `import` function

- Python, by default, only has access to a small number of built-in types and functions. The vast majority of functions are located in modules, and before a function can be accessed, the module which contains the function must be imported.
- For example, when using `ipython --pylab` (or any variants), a large number of modules are automatically imported, including NumPy and matplotlib.
- This is style of importing useful for learning and interactive use, but care is needed to make sure that the correct module is imported when designing more complex programs.
- `import` can be used in a variety of ways. The simplest is to use from module `import *` which imports all functions in module.
- This method of using import can dangerous since if you use it more than once, it is possible for functions to be hidden by later imports.
- A better method is to just import the required functions.
- This still places functions at the top level of the namespace, but can be used to avoid conflicts.

```
from pylab import log2 # Will import log2 only
from scipy import log10 # Will not import the log2 from SciPy
```

- The only difference between these two is that `import scipy` is implicitly calling `import scipy as scipy`.
- When this formof import is used, functions are used with the "as" name. For example, the load provided by NumPy is accessed using sp.log2, while the pylab load is `pl.log2` – and both can be used where appropriate.
- While this method is the most general, it does require slightly more typing.