```
'Output':
array([[2, 2, 2],
       [2, 2, 2],
       [2, 2, 2]])
# create a 3x3, empty uninitialized array
np.empty([3,3])
'Output':
array([[ -2.00000000e+000,  -2.00000000e+000,   2.47032823e-323],
       [  0.00000000e+000,   0.00000000e+000,   0.00000000e+000],
       [ -2.00000000e+000,  -1.73060571e-077,  -2.00000000e+000]])
# create a 4x4 identity matrix - i.e., a matrix with 1's on its diagonal
np.eye(4) # or np.identity(4)
'Output':
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Creating 3-D Arrays

Let’s construct a basic 3-D array.

```
# construct a 3-D array
my_3D = np.array([[
                    [2,4,6],
                    [8,10,12]
                   ],[
                    [1,2,3],
                    [7,9,11]
                   ]])
my_3D
'Output':
array([[[ 2,  4,  6],
        [ 8, 10, 12]],

       [[ 1,  2,  3],
        [ 7,  9, 11]]])
```

```
# check the number of dimensions
my_3D.ndim
'Output': 3
# get the shape of the 3-D array - this example has 2 pages, 2 rows and 3
columns: (p, r, c)
my_3D.shape
'Output': (2, 2, 3)
```

We can also create 3-D arrays with methods such as **ones**, **zeros**, **full**, and **empty** by passing the configuration for [page, row, columns] into the **shape** parameter of the methods. For example:

```
# create a 2-page, 3x3 array of ones
np.ones([2,3,3])
'Output':
array([[[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]]])
# create a 2-page, 3x3 array of zeros
np.zeros([2,3,3])
'Output':
array([[[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]],

       [[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]]])
```

Indexing/Slicing of Matrices

Let’s see some examples of indexing and slicing 2-D arrays. The concept extends nicely from doing the same with 1-D arrays.