```
# create an array of even numbers between 2 and 10
my_array = np.arange(2,11,2)
'Output': array([ 2,  4,  6,  8, 10])
# sum of array elements
np.sum(my_array) # or my_array.sum()
'Output': 30
# square root
np.sqrt(my_array)
'Output': array([ 1.41421356,  2.        ,  2.44948974,  2.82842712,
                  3.16227766])
# log
np.log(my_array)
'Output': array([ 0.69314718,  1.38629436,  1.79175947,  2.07944154,
                  2.30258509])
# exponent
np.exp(my_array)
'Output': array([  7.38905610e+00,   5.45981500e+01,   4.03428793e+02,
                   2.98095799e+03,   2.20264658e+04])
```

Higher-Dimensional Arrays

As we've seen earlier, the strength of NumPy is its ability to construct and manipulate n-dimensional arrays with highly optimized (i.e., vectorized) operations. Previously, we covered the creation of 1-D arrays (or vectors) in NumPy to get a feel of how NumPy works.

This section will now consider working with 2-D and 3-D arrays. 2-D arrays are ideal for storing data for analysis. Structured data is usually represented in a grid of rows and columns. And even when data is not necessarily represented in this format, it is often transformed into a tabular form before doing any data analytics or machine learning. Each column represents a feature or attribute and each row an observation.

Also, other data forms like images are adequately represented using 3-D arrays. A colored image is composed of $n \times n$ pixel intensity values with a color depth of three for the red, green, and blue (RGB) color profiles.

Creating 2-D Arrays (Matrices)

Let us construct a simple 2-D array.

```
# construct a 2-D array
my_2D = np.array([[2,4,6],
                    [8,10,12]])
my_2D
'Output':
array([[ 2,  4,  6],
       [ 8, 10, 12]])
# check the number of dimensions
my_2D.ndim
'Output': 2
# get the shape of the 2-D array - this example has 2 rows and
3 columns: (r, c)
my_2D.shape
'Output': (2, 3)
```

Let's explore common methods in practice for creating 2-D NumPy arrays, **which are also matrices**.

```
# create a 3x3 array of ones
np.ones([3,3])
'Output':
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
# create a 3x3 array of zeros
np.zeros([3,3])
'Output':
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
# create a 3x3 array of a particular scalar - full(shape, fill_value)
np.full([3,3], 2)
```