Advanced Python Demonstration: The Numerical Python Library (NumPy)

Now, we will be learning Numpy, a package widely used in the data science community which lets us work efficiently with arrays and matrices in Python.

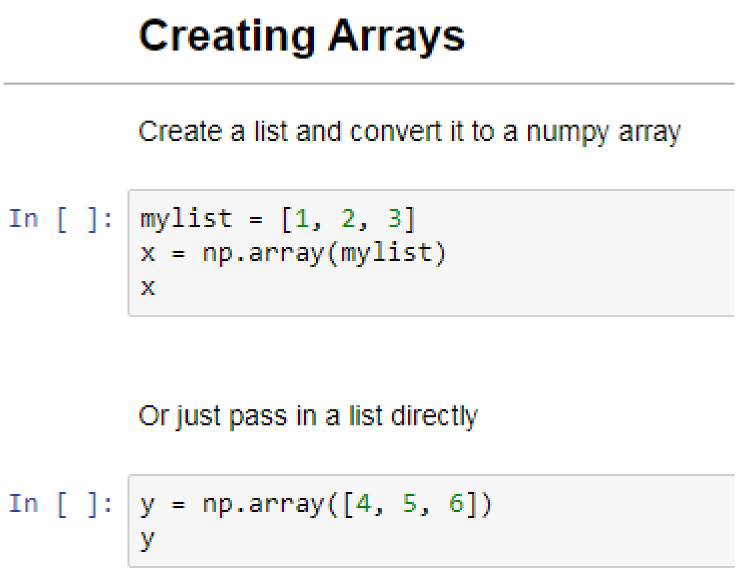
First, let's import Numpy as np.

```
In [1]: import numpy as np
```

This lets us use the shortcut np to refer to Numpy.

Now let's make our first array.
We can start by creating a list and converting it to an array.

So here's our first Numpy array.



Creating Arrays

Create a list and convert it to a numpy array

```
In [ ]: mylist = [1, 2, 3]
        x = np.array(mylist)
        x
```

Or just pass in a list directly

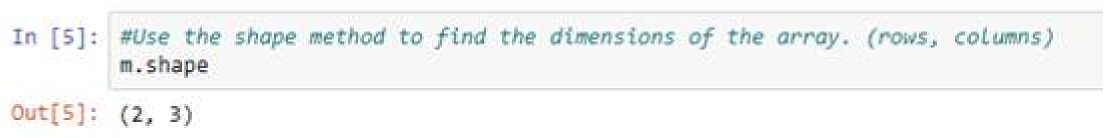
```
In [ ]: y = np.array([4, 5, 6])
        y
```

Now let's make multidimensional arrays by passing in a list of lists.

```
In [4]: #Pass in a list of lists to create a multidimensional array.
        m = np.array([[7, 8, 9], [10, 11, 12]])
        m

Out[4]: array([[ 7,  8,  9],
               [10, 11, 12]])
```

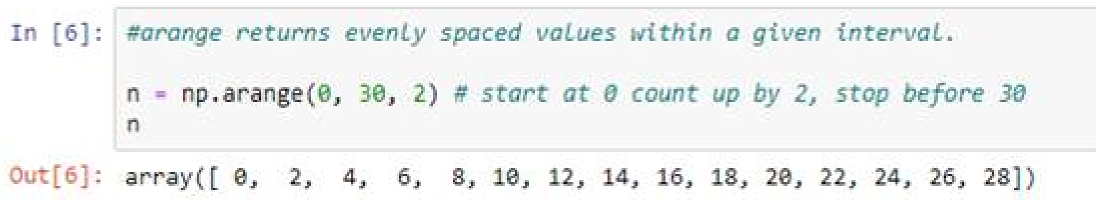
We passed in two lists with three elements each, and we get a two by three array.

We can check the dimensions by using the shape attribute.



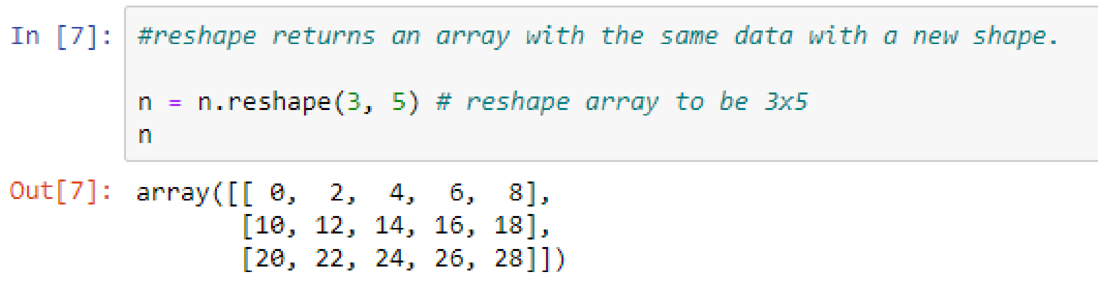
```
In [5]: #Use the shape method to find the dimensions of the array. (rows, columns)
        m.shape

Out[5]: (2, 3)
```

For the arange function, we pass in a start, a stop, and a step size, and it returns evenly spaced values within a given interval.



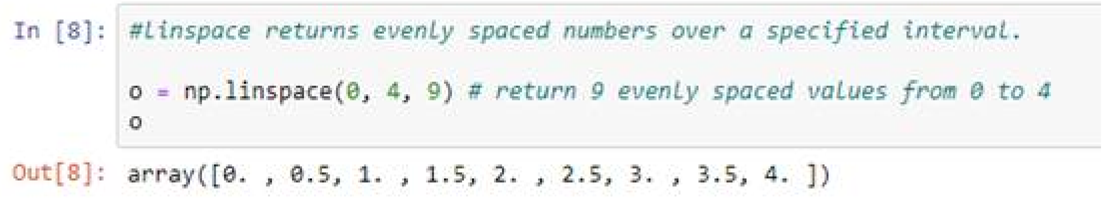
```
In [6]: #arange returns evenly spaced values within a given interval.

        n = np.arange(0, 30, 2) # start at 0 count up by 2, stop before 30
        n

Out[6]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

So suppose we wanted to convert this array of numbers to a three by five array.
We can use reshape to do that.



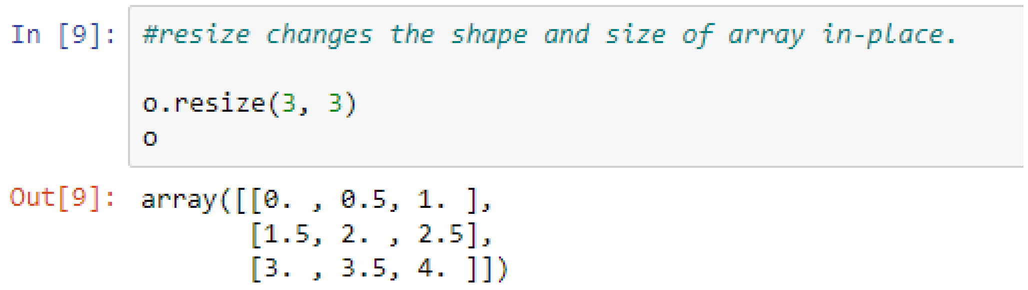
```
In [7]: #reshape returns an array with the same data with a new shape.

        n = n.reshape(3, 5) # reshape array to be 3x5
        n

Out[7]: array([[ 0,  2,  4,  6,  8],
               [10, 12, 14, 16, 18],
               [20, 22, 24, 26, 28]])
```

The linspace function is similar to arange, except we tell it how many numbers we want returned, and it will split up the interval accordingly.



```
In [8]: #linspace returns evenly spaced numbers over a specified interval.

        o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
        o

Out[8]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ])
```

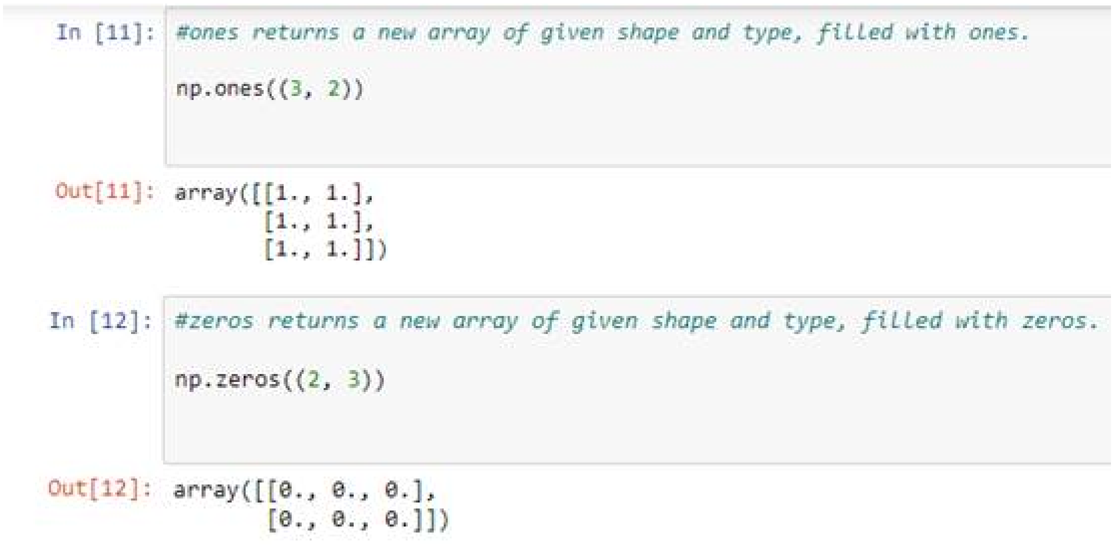
We can use resize to change the dimensions in place.



```
In [9]: #resize changes the shape and size of array in-place.

        o.resize(3, 3)
        o

Out[9]: array([[0. , 0.5, 1. ],
               [1.5, 2. , 2.5],
               [3. , 3.5, 4. ]])
```

Numpy also has several built-in functions and shortcuts for creating arrays.

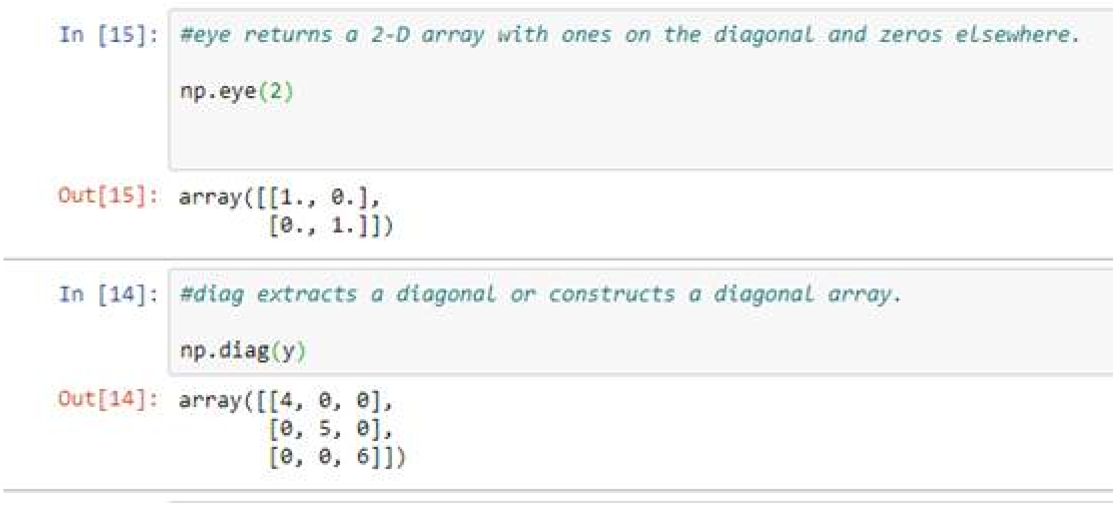
ones returns an array of ones, zeros an array of zeros.



```
In [11]: #ones returns a new array of given shape and type, filled with ones.

         np.ones((3, 2))

Out[11]: array([[1., 1.],
                [1., 1.],
                [1., 1.]])

In [12]: #zeros returns a new array of given shape and type, filled with zeros.

         np.zeros((2, 3))

Out[12]: array([[0., 0., 0.],
                [0., 0., 0.]])
```

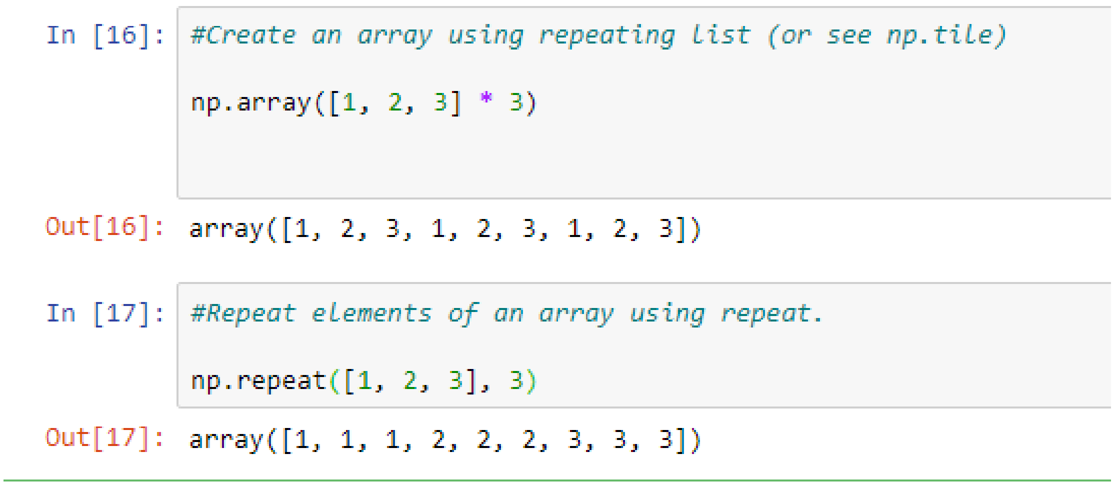
eye returns an array with ones on the diagonal and zeros everywhere else,

and diag constructs a diagonal array.



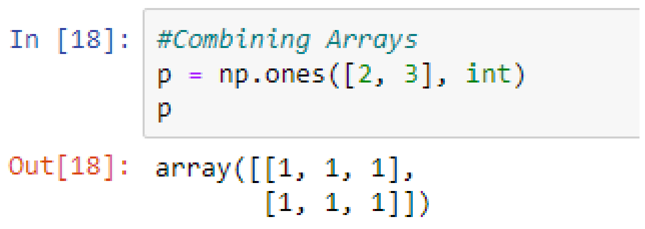
```
In [15]: #eye returns a 2-D array with ones on the diagonal and zeros elsewhere.

         np.eye(2)

Out[15]: array([[1., 0.],
                [0., 1.]])

In [14]: #diag extracts a diagonal or constructs a diagonal array.

         np.diag(y)

Out[14]: array([[4, 0, 0],
                [0, 5, 0],
                [0, 0, 6]])
```

To create an array with repeated values, we can pass in a repeated list, or we can use Numpy's repeat function.
Notice the difference between the two outputs.



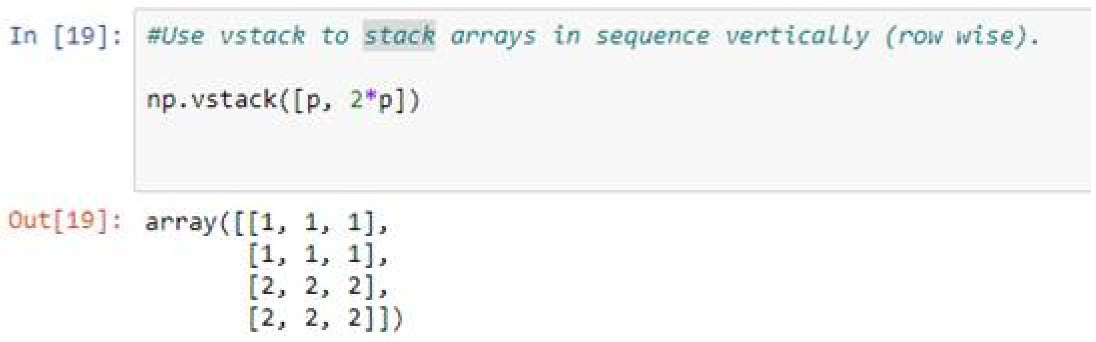
```
In [16]: #Create an array using repeating list (or see np.tile)

         np.array([1, 2, 3] * 3)

Out[16]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])

In [17]: #Repeat elements of an array using repeat.

         np.repeat([1, 2, 3], 3)

Out[17]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

We can also combine arrays to create new ones.



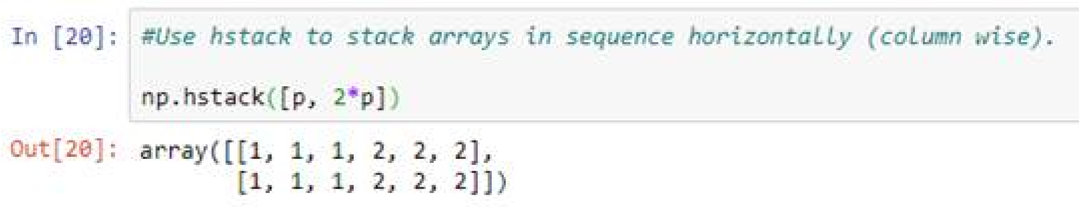
```
In [18]: #Combining Arrays
         p = np.ones([2, 3], int)
         p

Out[18]: array([[1, 1, 1],
                [1, 1, 1]])
```

Let's create a two by three array of ones and stack it vertically with itself, multiplied by 2.



```
In [19]: #Use vstack to stack arrays in sequence vertically (row wise).

         np.vstack([p, 2*p])

Out[19]: array([[1, 1, 1],
                [1, 1, 1],
                [2, 2, 2],
                [2, 2, 2]])
```

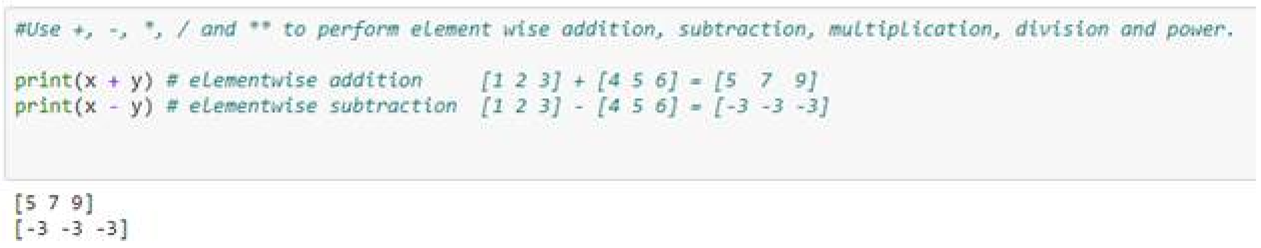
And here's the same thing, but stacking horizontally.



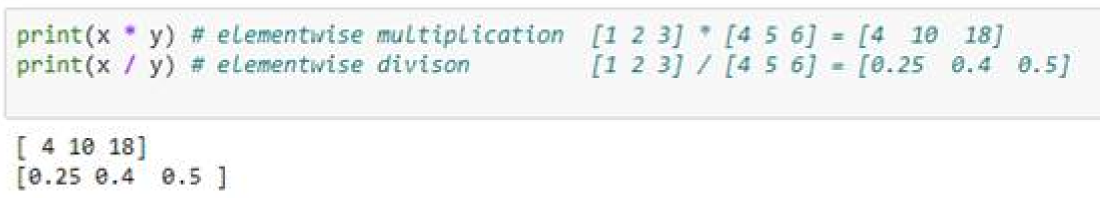
```
In [20]: #Use hstack to stack arrays in sequence horizontally (column wise).

         np.hstack([p, 2*p])

Out[20]: array([[1, 1, 1, 2, 2, 2],
                [1, 1, 1, 2, 2, 2]])
```

Now, let's look into some of the operations you can do with Numpy arrays.

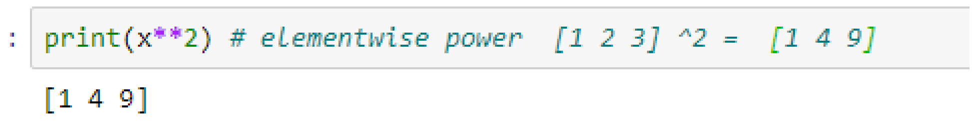
Performing element wise addition, subtraction, multiplication, and division is straightforward, as is raising all the numbers of an array to a power.



```
#Use +, -, *, / and ** to perform element wise addition, subtraction, multiplication, division and power.

print(x + y) # elementwise addition     [1 2 3] + [4 5 6] = [5  7  9]
print(x - y) # elementwise subtraction  [1 2 3] - [4 5 6] = [-3 -3 -3]

[5 7 9]
[-3 -3 -3]
```



```
print(x * y) # elementwise multiplication  [1 2 3] * [4 5 6] = [4  10  18]
print(x / y) # elementwise divison         [1 2 3] / [4 5 6] = [0.25  0.4  0.5]

[ 4 10 18]
[0.25 0.4  0.5 ]
```



```
: print(x**2) # elementwise power  [1 2 3] ^2 =  [1 4 9]

  [1 4 9]
```

For those familiar with linear algebra, the dot product can be done using the dot function.

Dot Product:

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = x_1 y_1 + x_2 y_2 + x_3 y_3$$

```
x.dot(y) # dot product  1*4 + 2*5 + 3*6
```

```
32
```

```
z = np.array([y, y**2])
print(len(z)) # number of rows of array
```

```
2
```

Let's create a new array using a previous array y and its squared values.

The shape of this array is two by three.

```
#Let's look at transposing arrays. Transposing permutes the dimensions of the array.

z = np.array([y, y**2])
z
```

```
array([[ 4,  5,  6],
       [16, 25, 36]])
```

We can also take the transpose of an array using the t method, which swaps the rows and columns.
The shape of the transposed array is three by two.

```
#The shape of array z is (2,3) before transposing.

z.shape
```

```
(2, 3)
```

```
#Use .T to get the transpose.

z.T
```

```
array([[ 4, 16],
       [ 5, 25],
       [ 6, 36]])
```

Using dtype, we can see what type of data the array has, and with astype, cast an array to a different type.

```
#Use .dtype to see the data type of the elements in the array.

z.dtype
```

```
dtype('int32')
```

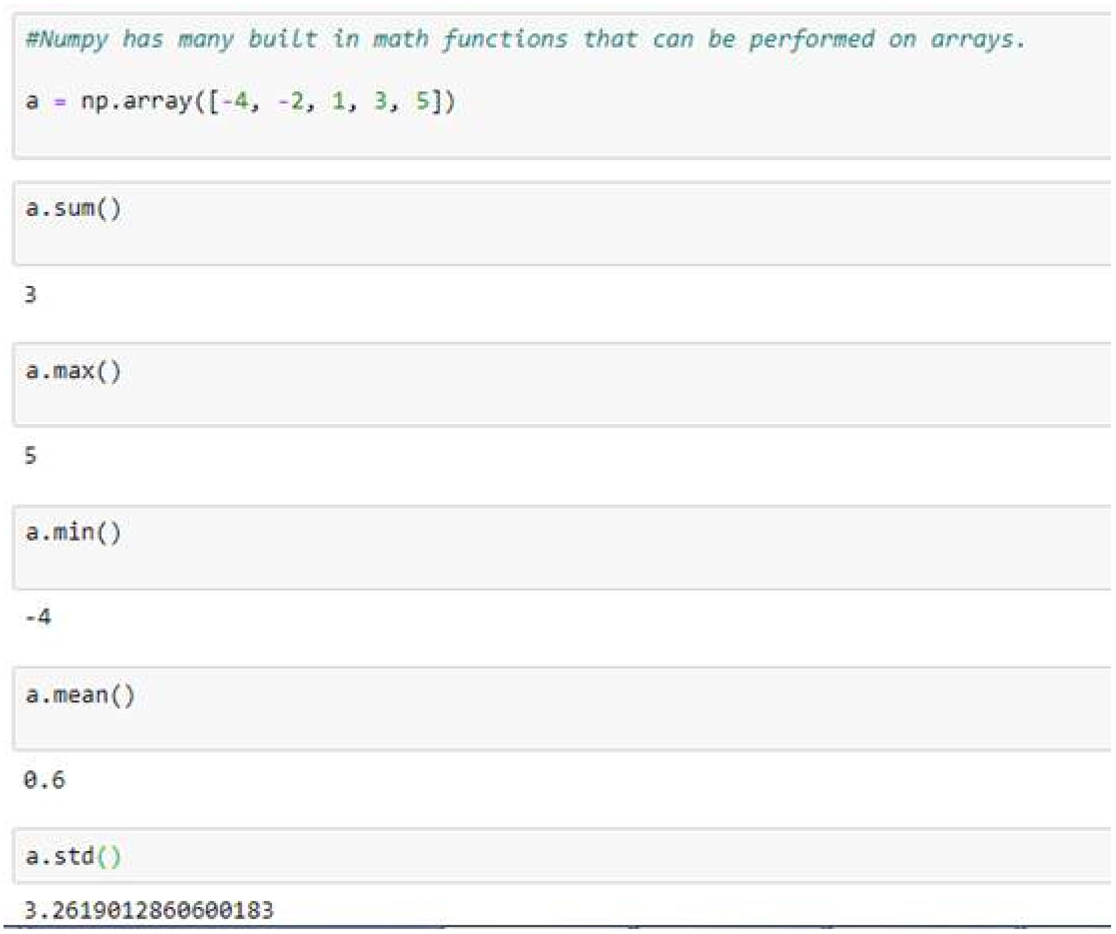
```
#Use .astype to cast to a specific type.

z = z.astype('f')
z.dtype
```

```
dtype('float32')
```

Math Functions

Numpy also has many useful math functions that we can use.
Let's look at a few commonly used ones.

Here's our new array a.



```
#Numpy has many built in math functions that can be performed on arrays.

a = np.array([-4, -2, 1, 3, 5])
```

```
a.sum()
```

```
3
```

```
a.max()
```

```
5
```

```
a.min()
```

```
-4
```

```
a.mean()
```

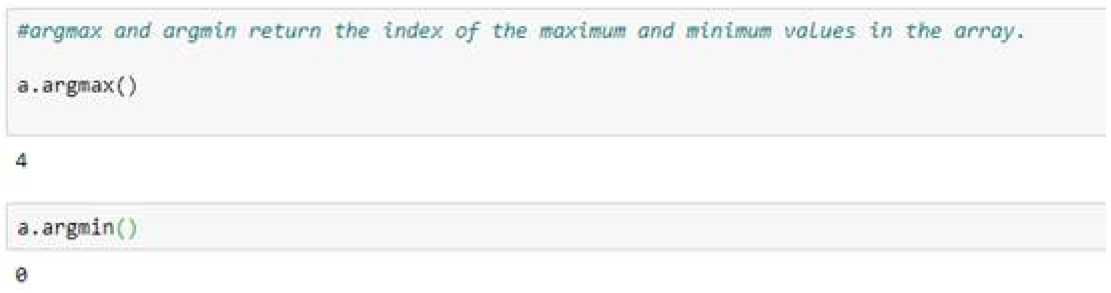
```
0.6
```

```
a.std()
```

```
3.2619012860600183
```

And we can look the sum of the values in the array, the maximum and minimum, Or the mean and standard deviation.

To find the index of a maximum or minimum value, we can use argmax and argmin.



```
#argmax and argmin return the index of the maximum and minimum values in the array.

a.argmax()
```

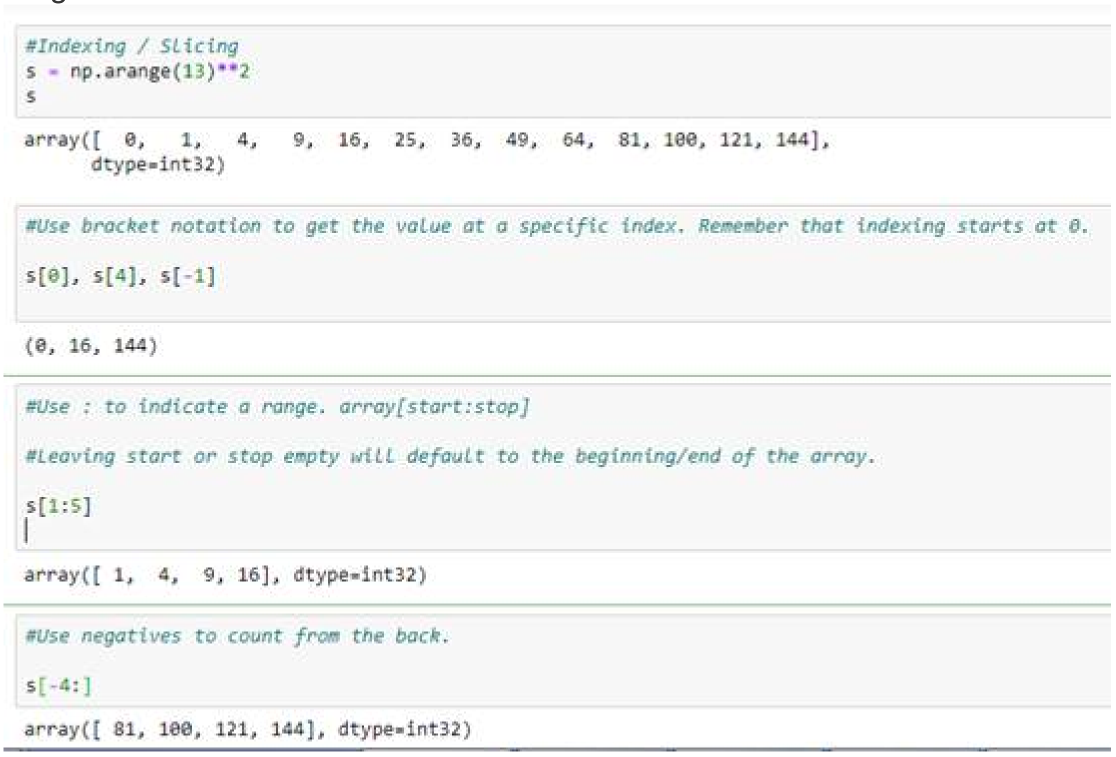
```
4
```

```
a.argmin()
```

```
0
```

Indexing / Slicing
Next, let's learn about indexing and slicing.

Let's create an array with the squares of 0 through 12.
We can use bracket notation to get the value at a particular index, and the colon notation to get a range.



```
#Indexing / Slicing
s = np.arange(13)**2
s
```

```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144],
      dtype=int32)
```

```
#Use bracket notation to get the value at a specific index. Remember that indexing starts at 0.

s[0], s[4], s[-1]
```

```
(0, 16, 144)
```

```
#Use : to indicate a range. array[start:stop]

#Leaving start or stop empty will default to the beginning/end of the array.

s[1:5]
```

```
array([ 1,  4,  9, 16], dtype=int32)
```

```
#Use negatives to count from the back.

s[-4:]
```

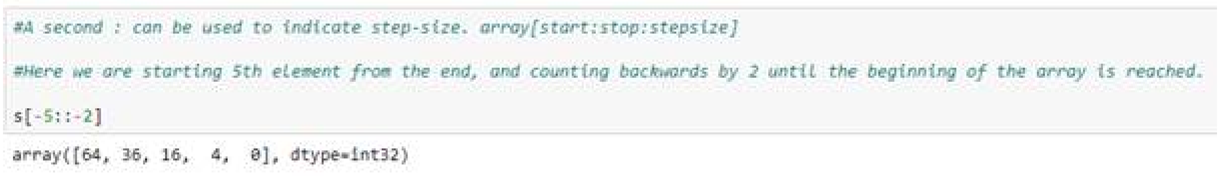
```
array([ 81, 100, 121, 144], dtype=int32)
```

The notation is start and stepsize.

Specifying the starting or ending index is not necessary.
And we can also use negatives to count back from the end of the array.
For our first example, let's take a look at the range starting from index one and stopping before index five.

Next, let's get a slice of the last four elements of the array.

And here, we're starting fifth from the end to the beginning of the array and counting backwards by two.



```
#A second : can be used to indicate step-size. array[start:stop:stepsize]

#Here we are starting 5th element from the end, and counting backwards by 2 until the beginning of the array is reached.

s[-5::-2]
```

```
array([64, 36, 16,  4,  0], dtype=int32)
```

Let's see how this extends to a two-dimensional array.

```
#Let's look at a multidimensional array.

r = np.arange(36)
r.resize((6, 6))
r
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

First, let's make a two dimensional array, 0 to 35.
We can get a specific value by using the comma notation.
Here's the value at the second row and second column.

```
#ue bracket notation to slice: array[row, column]

r[2, 2]
```

```
14
```

Now let's use colon notation to get a slice of the third row and columns three to six.

```
#and use : to select a range of rows or columns

r[3, 3:6]
```

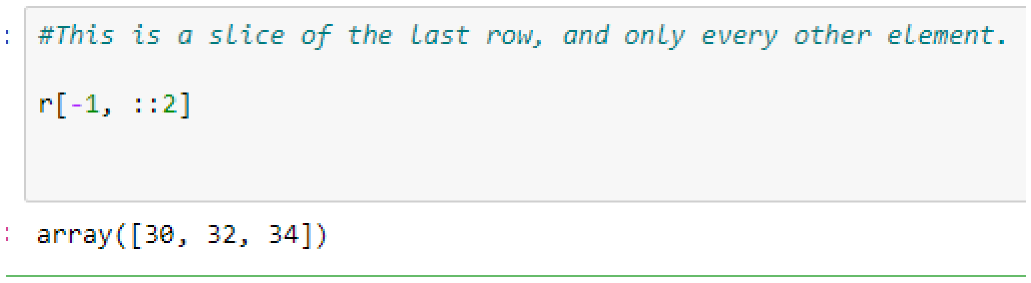
```
array([21, 22, 23])
```

We can also do something like get the first two rows and all of the columns except the last.

```
#Here we are selecting all the rows up to (and not including) row 2,
#and all the columns up to (and not including) the last column.

r[:2, :-1]
```

```
array([[ 0,  1,  2,  3,  4],
       [ 6,  7,  8,  9, 10]])
```

This is how we would select every second element from the last row.

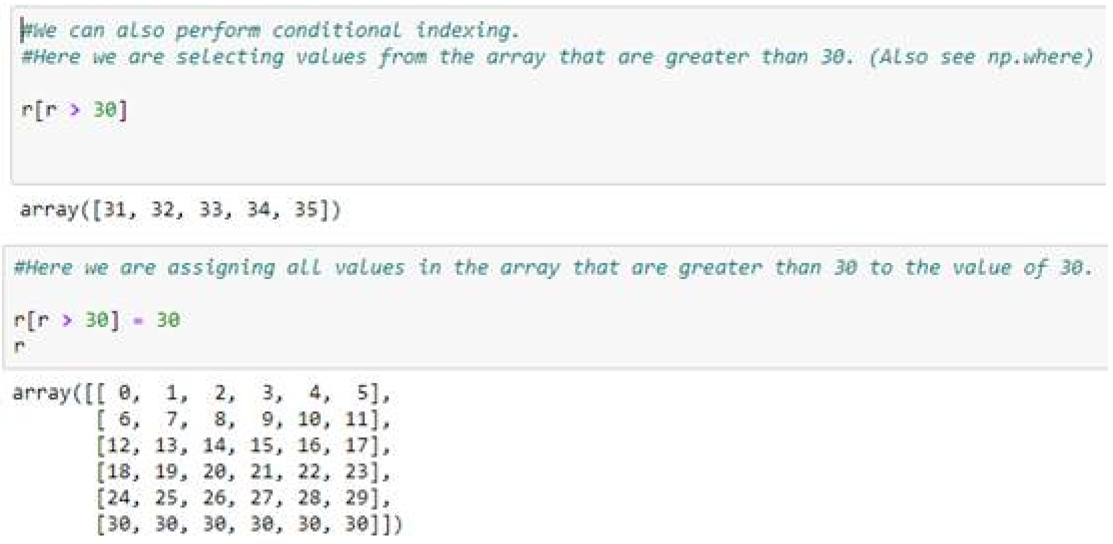


```
#This is a slice of the last row, and only every other element.

r[-1, ::2]
```

```
array([30, 32, 34])
```

We can also use the bracket operator to do conditional indexing and assignment.

For example, this will return an array that is the elements of our original array that are greater than 30.



```
#We can also perform conditional indexing.
#Here we are selecting values from the array that are greater than 30. (Also see np.where)

r[r > 30]
```

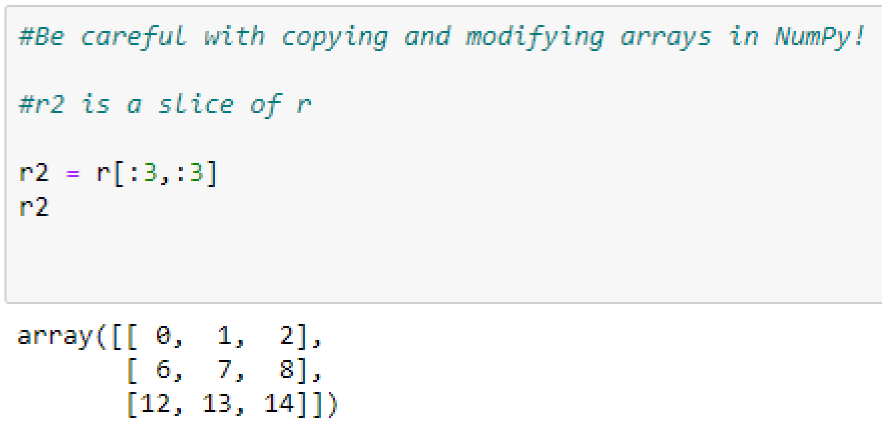
```
array([31, 32, 33, 34, 35])
```

```
#Here we are assigning all values in the array that are greater than 30 to the value of 30.

r[r > 30] = 30
r
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 30, 30, 30, 30, 30]])
```

Copying Data

Here, we're capping the maximum value of the elements in our array to 30.
Next, let's look at copying data in Numpy.
First, let's create a new array r2, which is a slice of the array r.



```
#Be careful with copying and modifying arrays in NumPy!

#r2 is a slice of r

r2 = r[:3,:3]
r2
```

```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [12, 13, 14]])
```

Now, let's set all the elements of this array to zero.

```
#Set this slice's values to zero ([:] selects the entire array)

r2[:] = 0
r2
```

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

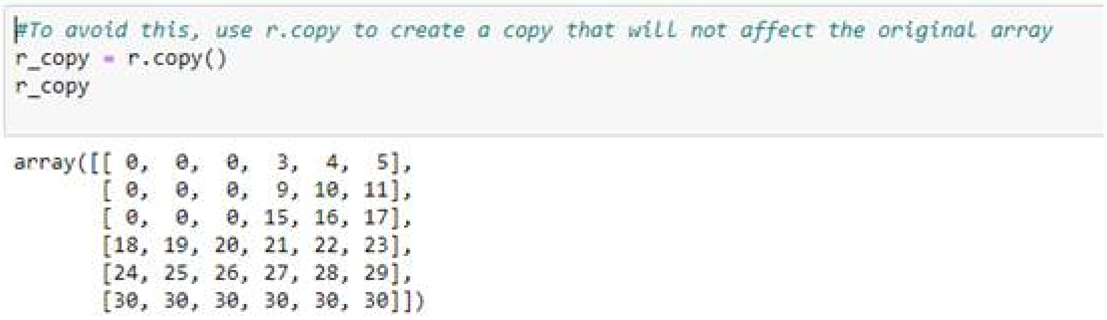
When we look at the original array r, we can see that the slice in r has also been changed.

So this is something to keep in mind and be careful about when working with Numpy arrays.

```
#r has also been changed!

r
```

```
array([[ 0,  0,  0,  3,  4,  5],
       [ 0,  0,  0,  9, 10, 11],
       [ 0,  0,  0, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 30, 30, 30, 30, 30]])
```

If we wish to create a copy of the r array that will not change the original array, we can use r_copy.



```
#To avoid this, use r.copy to create a copy that will not affect the original array
r_copy = r.copy()
r_copy
```

```
array([[ 0,  0,  0,  3,  4,  5],
       [ 0,  0,  0,  9, 10, 11],
       [ 0,  0,  0, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 30, 30, 30, 30, 30]])
```

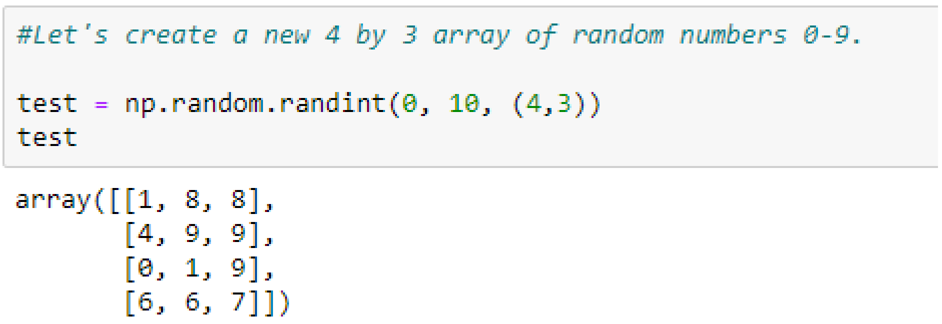
We can see that if we change the values of all the elements in r_copy to ten, the original array r remains unchanged.

```
#Now when r_copy is modified, r will not be changed.
r_copy[:] = 10
print(r_copy, '\n')
print(r)
```

```
[[10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]]

[[ 0  0  0  3  4  5]
 [ 0  0  0  9 10 11]
 [ 0  0  0 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 30 30 30 30 30]]
```

Iterating Over Arrays

Lastly, let's learn how to iterate over arrays.
First, let's create a four by three array of random numbers, from zero through nine



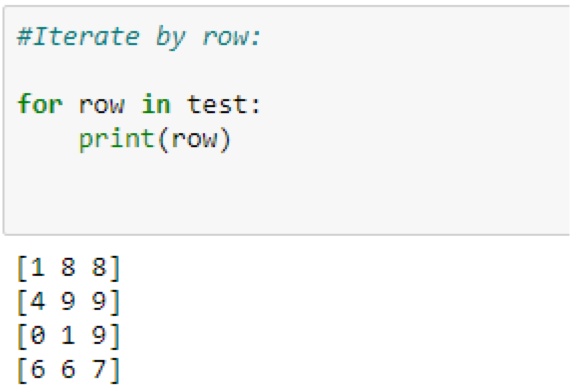
```
#Let's create a new 4 by 3 array of random numbers 0-9.

test = np.random.randint(0, 10, (4,3))
test
```

```
array([[1, 8, 8],
       [4, 9, 9],
       [0, 1, 9],
       [6, 6, 7]])
```

.

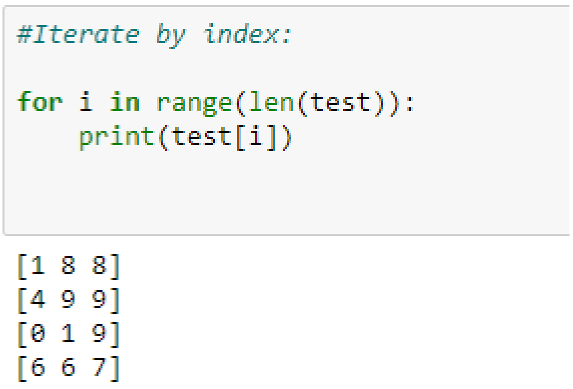
We can iterate by row by typing for row in test, for example.

We can iterate by row index by using the length function on test, which returns the number of rows.



```
#Iterate by row:

for row in test:
    print(row)
```

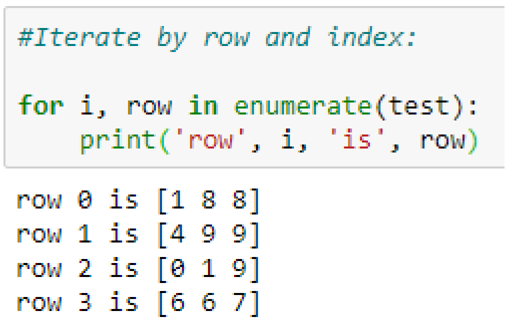
```
[1 8 8]
[4 9 9]
[0 1 9]
[6 6 7]
```



```
#Iterate by index:

for i in range(len(test)):
    print(test[i])
```

```
[1 8 8]
[4 9 9]
[0 1 9]
[6 6 7]
```

We can combine these two ways of iterating by using enumerate, which gives us the row and the index of the row.



```
#Iterate by row and index:

for i, row in enumerate(test):
    print('row', i, 'is', row)
```

```
row 0 is [1 8 8]
row 1 is [4 9 9]
row 2 is [0 1 9]
row 3 is [6 6 7]
```

Let's make a new array, test2.

If we wish to iterate through both arrays, we can use zip.

```
#Use zip to iterate over multiple iterables.

test2 = test**2
test2
```

```
array([[ 1, 64, 64],
       [16, 81, 81],
       [ 0,  1, 81],
       [36, 36, 49]], dtype=int32)
```

:
```
for i, j in zip(test, test2):
    print(i,'+',j,'=',i+j)
```

```
[1 8 8] + [ 1 64 64] = [ 2 72 72]
[4 9 9] + [16 81 81] = [20 90 90]
[0 1 9] + [ 0  1 81] = [ 0  2 90]
[6 6 7] + [36 36 49] = [42 42 56]
```

Numpy has a lot to offer.
So be sure to look at the documentation to find out about more great features.