Module 3 Part 2: Operations with NumPy Arrays

This module consists of 2 parts:

- **Part 1** - NumPy Basics
- **Part 2** - Operations with NumPy Arrays

Each part is provided in a separate notebook file. It is recommended that you follow the order of the notebooks.

Table of Contents

- Module 3 Part 2: Operations with NumPy Arrays
- Table of Contents
- Operations with NumPy Arrays
  - Operations between arrays and scalars
  - Arithmetic with NumPy arrays
    - Broadcasting
    - EXERCISE 3: Array mathematics
- Basic Indexing and Slicing
  - Indexing with slices
  - EXERCISE 4: Indexing and Slicing

Operations with NumPy Arrays

Operations between arrays and scalars

You can perform arithmetic operations between arrays of any shape and scalars. In the result of this operation, the values are propagated to each element of the array.

For example, let's say we need to multiply all elements of an array by 5:

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

# We will use an array of integers that we created in the previous section
```

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

Out[1]:
```
array([1, 2, 3, 4, 5])
```

In [2]:
```
# Multiply all elements of the array by 5

arr_int * 5
```

Out[2]:
```
array([ 5, 10, 15, 20, 25])
```

In [3]:
```
# Raise all elements of the array, to the power of 2

1 / arr_int ** 2
```

Out[3]:
```
array([1.        , 0.25      , 0.11111111, 0.0625    , 0.04      ])
```

Arithmetic with NumPy arrays

Any arithmetic operations between equal-size arrays applies the operation element-wise:

In [4]:
```
array1 = np.array([[5, 4, 6, 1], [2, 3, 8, 10]])
array2 = np.array([[10, 12, 5, 17], [22, 33, 88, 100]])
print(array1)
print(array2)
```

```
[[ 5  4  6  1]
 [ 2  3  8 10]]
[[ 10  12   5  17]
 [ 22  33  88 100]]
```

In [5]:
```
array1 + array2
```

Out[5]:
```
array([[ 15,  16,  11,  18],
       [ 24,  36,  96, 110]])
```

In [49]:
```
# Multiply arrays

array1 * array2
```

Out[49]:
```
array([[  50,   48,   30,   17],
       [  44,   99,  704, 1000]])
```

In [50]:
```
# Subtract two arrays

array1 - array2
```

Out[50]:
```
array([[ -5,  -8,   1, -16],
       [-20, -30, -80, -90]])
```

In [51]:
```
# Divide arrays

array1 / array2
```

```
Out[51]: array([[0.5       , 0.33333333, 1.2       , 0.05882353],
                [0.09090909, 0.09090909, 0.09090909, 0.1       ]])
```

```
In [52]: # Calculate the remainder of two arrays

         array1 % array2
```

```
Out[52]: array([[ 5,  4,  1,  1],
                [ 2,  3,  8, 10]])
```

As you can see, we can use `+`, `-`, `*`, `/` or `%` to add, subtract, multiply, divide or calculate the remainder of the arrays element-wise.

We can achieve the same result by using NumPy functions: `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()` and `np.remainder()`.

```
In [6]: np.add(array1,array2)
```

```
Out[6]: array([[ 15,  16,  11,  18],
               [ 24,  36,  96, 110]])
```

```
In [7]: np.multiply(array1,array2)
```

```
Out[7]: array([[  50,   48,   30,   17],
               [  44,   99,  704, 1000]])
```

```
In [8]: np.subtract(array1,array2)
```

```
Out[8]: array([[ -5,  -8,   1, -16],
               [-20, -30, -80, -90]])
```

```
In [9]: np.divide(array1,array2)
```

```
Out[9]: array([[0.5       , 0.33333333, 1.2       , 0.05882353],
               [0.09090909, 0.09090909, 0.09090909, 0.1       ]])
```

```
In [10]: np.remainder(array1,array2)
```

```
Out[10]: array([[ 5,  4,  1,  1],
                [ 2,  3,  8, 10]])
```

Broadcasting

The term **broadcasting** refers to how NumPy treats arrays with different shapes during arithmetic operations. Frequently, we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array. For example, suppose that we want to add a constant vector to each row of a matrix. The smaller array is "broadcast" across the larger array so that they have compatible shapes.

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when:

- they are equal, or
- one of them is 1

The size of the resulting array is the maximum size along each dimension of the input arrays.

Arrays do not need to have the same number of dimensions. Dimensions with size 1 are stretched or "copied" to match the other dimensions. For example, below you can see that both input arrays, `A` and `B`, have axes with length one and expand to a larger size during the broadcast operation:

```
A      (4-dimensional array):  8 x 1 x 6 x 1
B      (3-dimensional array):      7 x 1 x 5
Result (4-dimensional array):  8 x 7 x 6 x 5
```

More examples:

```
A      (2-dim array):  5 x 4
B      (1-dim array):      4
Result (2-dim array):  5 x 4


A      (3-dim array):  15 x 3 x 5
B      (3-dim array):  15 x 1 x 5
Result (3-dim array):  15 x 3 x 5


A      (3-dim array):  15 x 3 x 5
B      (2-dim array):       3 x 1
Result (3-dim array):  15 x 3 x 5
```

Here is an example of broadcasting :

In [58]:
```
x1 = np.arange(4)        # create an array of integers from 0 to 3
x2 = x1.reshape(4,1)     # reshape() function allows us to change the shape of the
                         # the resulting array will be a vector, 4 rows and 1 colum
x2
```

Out[58]:
```
array([[0],
       [1],
       [2],
       [3]])
```

In [59]:
```
y = np.ones(5)          # creating an array of 1s, 5-element array
y
```

Out[59]:
```
array([1., 1., 1., 1., 1.])
```

In [60]:
```
x2.shape
```

Out[60]:
```
(4, 1)
```

In [61]:
```
y.shape
```

Out[61]:
```
(5,)
```

In [62]:
```
xy = x2 + y
xy
```

Out[62]:
```
array([[1., 1., 1., 1., 1.],
       [2., 2., 2., 2., 2.],
       [3., 3., 3., 3., 3.],
       [4., 4., 4., 4., 4.]])
```

In [63]:
```
xy.shape
```

Out[63]:
```
(4, 5)
```

In [64]:
```
# An array of 1s, 3 x 4

z = np.ones((3,4))
z
```

Out[64]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [65]:
```
x1 + z
```

Out[65]:
```
array([[1., 2., 3., 4.],
       [1., 2., 3., 4.],
       [1., 2., 3., 4.]])
```

If arrays are of the same size, we can compare them. The result of the comparison is a boolean array:

In [66]:
```
array1 = np.array([[5, 4, 6, 1], [2, 3, 8, 10]])
array1
```

Out[66]:
```
array([[ 5,  4,  6,  1],
       [ 2,  3,  8, 10]])
```

In [67]:
```
array2 = np.array([[0., 4., 1., 8.], [7., 2., 35, 12.]])
array2
```

Out[67]:
```
array([[ 0.,  4.,  1.,  8.],
       [ 7.,  2., 35., 12.]])
```

In [68]:
```
array2 > array1
```

Out[68]:
```
array([[False, False, False,  True],
       [ True, False,  True,  True]])
```

EXERCISE 3: Array mathematics

1). Create 2 arrays:

- Array `x` : array of 1s with dimensions 3 x 4,
- Array `y` : array of random numbers with dimensions 5 x 1 x 4

In [16]:
```
# type your code here

array_x = np.ones((3,4))
array_x
```

Out[16]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [19]:
```
array_y = np.random.random((5,1,4))
array_y
```

Out[19]:
```
array([[[0.01965301, 0.08568828, 0.27874257, 0.94697773]],

       [[0.1380056 , 0.07498052, 0.8357818 , 0.23230075]],

       [[0.16322056, 0.28295324, 0.74728177, 0.81758345]],

       [[0.23928793, 0.85687857, 0.42120051, 0.24821486]],

       [[0.70332014, 0.65773476, 0.38627052, 0.1334914 ]]])
```

2). Add `x` and `y`

In [20]:
```
# type your code here

array_xy = array_x + array_y
array_xy
```

Out[20]:
```
array([[[1.01965301, 1.08568828, 1.27874257, 1.94697773],
        [1.01965301, 1.08568828, 1.27874257, 1.94697773],
        [1.01965301, 1.08568828, 1.27874257, 1.94697773]],

       [[1.1380056 , 1.07498052, 1.8357818 , 1.23230075],
        [1.1380056 , 1.07498052, 1.8357818 , 1.23230075],
        [1.1380056 , 1.07498052, 1.8357818 , 1.23230075]],

       [[1.16322056, 1.28295324, 1.74728177, 1.81758345],
        [1.16322056, 1.28295324, 1.74728177, 1.81758345],
        [1.16322056, 1.28295324, 1.74728177, 1.81758345]],

       [[1.23928793, 1.85687857, 1.42120051, 1.24821486],
        [1.23928793, 1.85687857, 1.42120051, 1.24821486],
        [1.23928793, 1.85687857, 1.42120051, 1.24821486]],

       [[1.70332014, 1.65773476, 1.38627052, 1.1334914 ],
        [1.70332014, 1.65773476, 1.38627052, 1.1334914 ],
        [1.70332014, 1.65773476, 1.38627052, 1.1334914 ]]])
```

3). Subtract `x` and `y`

```
In [21]: # type your code here

         array_xy = array_x - array_y
         array_xy
```

```
Out[21]: array([[[0.98034699, 0.91431172, 0.72125743, 0.05302227],
                 [0.98034699, 0.91431172, 0.72125743, 0.05302227],
                 [0.98034699, 0.91431172, 0.72125743, 0.05302227]],

                [[0.8619944 , 0.92501948, 0.1642182 , 0.76769925],
                 [0.8619944 , 0.92501948, 0.1642182 , 0.76769925],
                 [0.8619944 , 0.92501948, 0.1642182 , 0.76769925]],

                [[0.83677944, 0.71704676, 0.25271823, 0.18241655],
                 [0.83677944, 0.71704676, 0.25271823, 0.18241655],
                 [0.83677944, 0.71704676, 0.25271823, 0.18241655]],

                [[0.76071207, 0.14312143, 0.57879949, 0.75178514],
                 [0.76071207, 0.14312143, 0.57879949, 0.75178514],
                 [0.76071207, 0.14312143, 0.57879949, 0.75178514]],

                [[0.29667986, 0.34226524, 0.61372948, 0.8665086 ],
                 [0.29667986, 0.34226524, 0.61372948, 0.8665086 ],
                 [0.29667986, 0.34226524, 0.61372948, 0.8665086 ]]])
```

4). Multiply `x` and `y`

```
In [22]: # type your code here

         array_xy = array_x * array_y
         array_xy
```

```
Out[22]: array([[[0.01965301, 0.08568828, 0.27874257, 0.94697773],
                 [0.01965301, 0.08568828, 0.27874257, 0.94697773],
                 [0.01965301, 0.08568828, 0.27874257, 0.94697773]],

                [[0.1380056 , 0.07498052, 0.8357818 , 0.23230075],
                 [0.1380056 , 0.07498052, 0.8357818 , 0.23230075],
                 [0.1380056 , 0.07498052, 0.8357818 , 0.23230075]],

                [[0.16322056, 0.28295324, 0.74728177, 0.81758345],
                 [0.16322056, 0.28295324, 0.74728177, 0.81758345],
                 [0.16322056, 0.28295324, 0.74728177, 0.81758345]],

                [[0.23928793, 0.85687857, 0.42120051, 0.24821486],
                 [0.23928793, 0.85687857, 0.42120051, 0.24821486],
                 [0.23928793, 0.85687857, 0.42120051, 0.24821486]],

                [[0.70332014, 0.65773476, 0.38627052, 0.1334914 ],
                 [0.70332014, 0.65773476, 0.38627052, 0.1334914 ],
                 [0.70332014, 0.65773476, 0.38627052, 0.1334914 ]]])
```

5). Divide `x` and `y`

In [24]:
```
# type your code here

array_xy = array_x / array_y
array_xy
```

Out[24]:
```
array([[[50.8827955 , 11.67020718,  3.58753962,  1.05599104],
        [50.8827955 , 11.67020718,  3.58753962,  1.05599104],
        [50.8827955 , 11.67020718,  3.58753962,  1.05599104]],

       [[ 7.24608264, 13.33679681,  1.19648454,  4.30476446],
        [ 7.24608264, 13.33679681,  1.19648454,  4.30476446],
        [ 7.24608264, 13.33679681,  1.19648454,  4.30476446]],

       [[ 6.1266791 ,  3.53415288,  1.33818331,  1.22311674],
        [ 6.1266791 ,  3.53415288,  1.33818331,  1.22311674],
        [ 6.1266791 ,  3.53415288,  1.33818331,  1.22311674]],

       [[ 4.17906573,  1.1670265 ,  2.37416617,  4.02876763],
        [ 4.17906573,  1.1670265 ,  2.37416617,  4.02876763],
        [ 4.17906573,  1.1670265 ,  2.37416617,  4.02876763]],

       [[ 1.42182761,  1.5203697 ,  2.58885922,  7.4911195 ],
        [ 1.42182761,  1.5203697 ,  2.58885922,  7.4911195 ],
        [ 1.42182761,  1.5203697 ,  2.58885922,  7.4911195 ]]])
```

In [74]:
```
# 1. Solution:

x = np.ones((3,4))
y = np.random.random((5,1,4))
```

In [75]:
```
x
```

Out[75]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [76]:
```
y
```

Out[76]:
```
array([[[0.99376807, 0.43860704, 0.69574599, 0.63777847]],

       [[0.25921884, 0.23312251, 0.98129715, 0.4875043 ]],

       [[0.96686017, 0.83122958, 0.97150656, 0.23935174]],

       [[0.33386857, 0.20340092, 0.66515148, 0.18836919]],

       [[0.4214884 , 0.90991346, 0.78864797, 0.51009636]]])
```

In [77]:
```
# 2. Solution
print(x + y)
# or
print(np.add(x, y))
```

```
[[[1.99376807 1.43860704 1.69574599 1.63777847]
  [1.99376807 1.43860704 1.69574599 1.63777847]
  [1.99376807 1.43860704 1.69574599 1.63777847]]

 [[1.25921884 1.23312251 1.98129715 1.4875043 ]
  [1.25921884 1.23312251 1.98129715 1.4875043 ]
  [1.25921884 1.23312251 1.98129715 1.4875043 ]]

 [[1.96686017 1.83122958 1.97150656 1.23935174]
  [1.96686017 1.83122958 1.97150656 1.23935174]
  [1.96686017 1.83122958 1.97150656 1.23935174]]

 [[1.33386857 1.20340092 1.66515148 1.18836919]
  [1.33386857 1.20340092 1.66515148 1.18836919]
  [1.33386857 1.20340092 1.66515148 1.18836919]]

 [[1.4214884  1.90991346 1.78864797 1.51009636]
  [1.4214884  1.90991346 1.78864797 1.51009636]
  [1.4214884  1.90991346 1.78864797 1.51009636]]]
[[[1.99376807 1.43860704 1.69574599 1.63777847]
  [1.99376807 1.43860704 1.69574599 1.63777847]
  [1.99376807 1.43860704 1.69574599 1.63777847]]

 [[1.25921884 1.23312251 1.98129715 1.4875043 ]
  [1.25921884 1.23312251 1.98129715 1.4875043 ]
  [1.25921884 1.23312251 1.98129715 1.4875043 ]]

 [[1.96686017 1.83122958 1.97150656 1.23935174]
  [1.96686017 1.83122958 1.97150656 1.23935174]
  [1.96686017 1.83122958 1.97150656 1.23935174]]

 [[1.33386857 1.20340092 1.66515148 1.18836919]
  [1.33386857 1.20340092 1.66515148 1.18836919]
  [1.33386857 1.20340092 1.66515148 1.18836919]]

 [[1.4214884  1.90991346 1.78864797 1.51009636]
  [1.4214884  1.90991346 1.78864797 1.51009636]
  [1.4214884  1.90991346 1.78864797 1.51009636]]]
```

In [78]:

```
# 3. Solution:
print(x - y)
# or
print(np.subtract(x, y))
```

```
[[[0.00623193 0.56139296 0.30425401 0.36222153]
  [0.00623193 0.56139296 0.30425401 0.36222153]
  [0.00623193 0.56139296 0.30425401 0.36222153]]

 [[0.74078116 0.76687749 0.01870285 0.5124957 ]
  [0.74078116 0.76687749 0.01870285 0.5124957 ]
  [0.74078116 0.76687749 0.01870285 0.5124957 ]]

 [[0.03313983 0.16877042 0.02849344 0.76064826]
  [0.03313983 0.16877042 0.02849344 0.76064826]
  [0.03313983 0.16877042 0.02849344 0.76064826]]

 [[0.66613143 0.79659908 0.33484852 0.81163081]
  [0.66613143 0.79659908 0.33484852 0.81163081]
  [0.66613143 0.79659908 0.33484852 0.81163081]]

 [[0.5785116  0.09008654 0.21135203 0.48990364]
  [0.5785116  0.09008654 0.21135203 0.48990364]
  [0.5785116  0.09008654 0.21135203 0.48990364]]]
[[[0.00623193 0.56139296 0.30425401 0.36222153]
  [0.00623193 0.56139296 0.30425401 0.36222153]
  [0.00623193 0.56139296 0.30425401 0.36222153]]

 [[0.74078116 0.76687749 0.01870285 0.5124957 ]
  [0.74078116 0.76687749 0.01870285 0.5124957 ]
  [0.74078116 0.76687749 0.01870285 0.5124957 ]]

 [[0.03313983 0.16877042 0.02849344 0.76064826]
  [0.03313983 0.16877042 0.02849344 0.76064826]
  [0.03313983 0.16877042 0.02849344 0.76064826]]

 [[0.66613143 0.79659908 0.33484852 0.81163081]
  [0.66613143 0.79659908 0.33484852 0.81163081]
  [0.66613143 0.79659908 0.33484852 0.81163081]]

 [[0.5785116  0.09008654 0.21135203 0.48990364]
  [0.5785116  0.09008654 0.21135203 0.48990364]
  [0.5785116  0.09008654 0.21135203 0.48990364]]]
```

In [79]:

```python
# 4. Solution:
print(x * y)
# or
print(np.multiply(x, y))
```

```
[[[0.99376807 0.43860704 0.69574599 0.63777847]
  [0.99376807 0.43860704 0.69574599 0.63777847]
  [0.99376807 0.43860704 0.69574599 0.63777847]]

 [[0.25921884 0.23312251 0.98129715 0.4875043 ]
  [0.25921884 0.23312251 0.98129715 0.4875043 ]
  [0.25921884 0.23312251 0.98129715 0.4875043 ]]

 [[0.96686017 0.83122958 0.97150656 0.23935174]
  [0.96686017 0.83122958 0.97150656 0.23935174]
  [0.96686017 0.83122958 0.97150656 0.23935174]]

 [[0.33386857 0.20340092 0.66515148 0.18836919]
  [0.33386857 0.20340092 0.66515148 0.18836919]
  [0.33386857 0.20340092 0.66515148 0.18836919]]

 [[0.4214884  0.90991346 0.78864797 0.51009636]
  [0.4214884  0.90991346 0.78864797 0.51009636]
  [0.4214884  0.90991346 0.78864797 0.51009636]]]
[[[0.99376807 0.43860704 0.69574599 0.63777847]
  [0.99376807 0.43860704 0.69574599 0.63777847]
  [0.99376807 0.43860704 0.69574599 0.63777847]]

 [[0.25921884 0.23312251 0.98129715 0.4875043 ]
  [0.25921884 0.23312251 0.98129715 0.4875043 ]
  [0.25921884 0.23312251 0.98129715 0.4875043 ]]

 [[0.96686017 0.83122958 0.97150656 0.23935174]
  [0.96686017 0.83122958 0.97150656 0.23935174]
  [0.96686017 0.83122958 0.97150656 0.23935174]]

 [[0.33386857 0.20340092 0.66515148 0.18836919]
  [0.33386857 0.20340092 0.66515148 0.18836919]
  [0.33386857 0.20340092 0.66515148 0.18836919]]

 [[0.4214884  0.90991346 0.78864797 0.51009636]
  [0.4214884  0.90991346 0.78864797 0.51009636]
  [0.4214884  0.90991346 0.78864797 0.51009636]]]
```

In [80]:

```
# 5. Solution
print(x / y)
# or
print(np.divide(x, y))
```

```
[[[1.00627101 2.27994519 1.43730617 1.56794254]
  [1.00627101 2.27994519 1.43730617 1.56794254]
  [1.00627101 2.27994519 1.43730617 1.56794254]]

 [[3.85774431 4.28959004 1.01905931 2.05126395]
  [3.85774431 4.28959004 1.01905931 2.05126395]
  [3.85774431 4.28959004 1.01905931 2.05126395]]

 [[1.03427572 1.20303707 1.02932913 4.17795161]
  [1.03427572 1.20303707 1.02932913 4.17795161]
  [1.03427572 1.20303707 1.02932913 4.17795161]]

 [[2.99519063 4.91639853 1.50341693 5.30872369]
  [2.99519063 4.91639853 1.50341693 5.30872369]
  [2.99519063 4.91639853 1.50341693 5.30872369]]

 [[2.37254451 1.09900561 1.26799287 1.9604139 ]
  [2.37254451 1.09900561 1.26799287 1.9604139 ]
  [2.37254451 1.09900561 1.26799287 1.9604139 ]]]
[[[1.00627101 2.27994519 1.43730617 1.56794254]
  [1.00627101 2.27994519 1.43730617 1.56794254]
  [1.00627101 2.27994519 1.43730617 1.56794254]]

 [[3.85774431 4.28959004 1.01905931 2.05126395]
  [3.85774431 4.28959004 1.01905931 2.05126395]
  [3.85774431 4.28959004 1.01905931 2.05126395]]

 [[1.03427572 1.20303707 1.02932913 4.17795161]
  [1.03427572 1.20303707 1.02932913 4.17795161]
  [1.03427572 1.20303707 1.02932913 4.17795161]]

 [[2.99519063 4.91639853 1.50341693 5.30872369]
  [2.99519063 4.91639853 1.50341693 5.30872369]
  [2.99519063 4.91639853 1.50341693 5.30872369]]

 [[2.37254451 1.09900561 1.26799287 1.9604139 ]
  [2.37254451 1.09900561 1.26799287 1.9604139 ]
  [2.37254451 1.09900561 1.26799287 1.9604139 ]]]
```

Basic Indexing and Slicing

One-dimensional arrays can be indexed, sliced and iterated over to select a subset of elements of NumPy arrays, similar to lists and other Python sequences. NumPy arrays are indexed like a list, so the index starts with zero.

In [25]:
```
a = np.arange(2,10)
a
```

Out[25]:
```
array([2, 3, 4, 5, 6, 7, 8, 9])
```

In [82]:
```
a[3]     # this index points to the 4th element of the array
```

```
Out[82]: 5
```

```
In [83]: a[1:5]    # elements with indexes 1 to 4, the end index is exclusive
```

```
Out[83]: array([3, 4, 5, 6])
```

A slicing operation creates a **view** on the original array, the data from the original array is not copied in memory. This also means that any changes to the elements of the slice (view) are done to the oiginal array. Let's illustrate:

```
In [84]: a_view = a[1:5]
         a_view[2] = 12      # change the value of the 3rd element of a view to 12
         a_view
```

```
Out[84]: array([ 3,  4, 12,  6])
```

The 3rd element of a view `a_view` is the 4th element of an array `a`:

```
In [85]: a
```

```
Out[85]: array([ 2,  3,  4, 12,  6,  7,  8,  9])
```

As you can see, the value of the 4th element of the original array `a` changed to `12` as well.

We might need to change all elements of a view. For this operation, we will use slice, `[:]`.

```
In [86]: a_view[:] = 345
         a
```

```
Out[86]: array([  2, 345, 345, 345, 345,   7,   8,   9])
```

For **multidimensional** arrays, indexes are tuples of integers.

For example, to select an index of a two-dimensional array:

```
In [87]: a_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         a_2d
```

```
Out[87]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [88]: # Indexing:

         a_2d[2]
```

```
Out[88]: array([7, 8, 9])
```

To select an individual element of a 2-dimensional array, we need to use a list of indices. In 2-dimensional arrays, the first dimension corresponds to rows, the second to columns:

In [89]:
```
a_2d[0,1]
```

Out[89]:
```
2
```

Indexing **multidimensional** arrays:

In [90]:
```
# Creating 2x2x3 array:

a_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
a_3d
```

Out[90]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

If we simply provide one index, `a_3d[0]`, the result will be a lower-dimensional `ndarray` of all the data along the higher dimensions, which is a `2 × 3` array:

In [91]:
```
a_3d[0]
```

Out[91]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

If 2 indices are specified `a_3d[1, 1]`, the result is a 1-dimensional array of all values where indices start with (1, 1):

In [92]:
```
a_3d[1, 1]
```

Out[92]:
```
array([10, 11, 12])
```

Indexing with slices

Slicing 2-dimensional arrays is slightly different than slicing 1-dimensional arrays. In a 2-dimensional array, a slice selects elements along the axis.

In [93]:
```
# Using 2-dimensional array from the previous section

a_2d
```

Out[93]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

The following line will select the first 2 rows — the rows with indices 0 and 1 — along axis = 0 (rows):

In [94]:
```
a_2d[:2]
```

```
Out[94]: array([[1, 2, 3],
                [4, 5, 6]])
```

To pass multiple slices:

```
In [95]: a_2d[:1, 1:]
```

```
Out[95]: array([[2, 3]])
```

If integer indexes and slices are mixed, the lower dimensional slices are selected. In the next example, we are selecting the 3rd row and only 1st column which in this particular example will return only 1 element:

```
In [96]: a_2d[2, :1]
```

```
Out[96]: array([7])
```

`[:]` selects an entire axis:

```
In [97]: a_2d[:, :2]
```

```
Out[97]: array([[1, 2],
                [4, 5],
                [7, 8]])
```

EXERCISE 4: Indexing and Slicing

1). Create a 2-dimensional 5x10 array of integers from 1 to 50 (inclusive).

HINT: You can create a one-dimensional array usingthe `arange()` function and change the shape of the array using the `reshape()` function introduced in the **Broadcasting** section above.

```
In [98]: # Type your code here:

         array5by10 = None
```

```
In [ ]:
```

2). Select the second row of the array.

```
In [ ]:
```

3). Select the second element from each row of the array. The result will be a one-dimensional array of 5 elements.

```
In [ ]:
```

4). Select numbers 16, 17 and 18 from the array.

In [ ]:

```

```

5). Update the fifth element in the third row of the array by replacing number 25 with 255.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [99]:

```
# Solution:
# 1). Creating an array

array5by10 = np.arange(1, 51).reshape(5,10)
array5by10
```

Out[99]:

```
array([[ 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, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]])
```

In [100…

```
# 2). Second row of the array

array5by10[1]
```

Out[100…

```
array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

In [101…

```
# 3). Second element from each row of the array will output a one-dimensional array

array5by10[:,1]
```

Out[101…

```
array([ 2, 12, 22, 32, 42])
```

In [102…

```
# 4). Selecting numbers 16, 17 and 18 from the array
# These numbers are in the second row (index = 1), their indices within the row are
# but we need to remember that the last index in the slice is not included

array5by10[1,5:8]
```

Out[102…

```
array([16, 17, 18])
```

In [103…

```
# 5). First, let's make sure we have the right index for number 25

array5by10[2,4]
```

Out[103…

```
25
```

In [104…

```
# Updating the element

array5by10[2,4] = 255
array5by10
```

Out[104…

```
array([[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10],
       [ 11,  12,  13,  14,  15,  16,  17,  18,  19,  20],
       [ 21,  22,  23,  24, 255,  26,  27,  28,  29,  30],
       [ 31,  32,  33,  34,  35,  36,  37,  38,  39,  40],
       [ 41,  42,  43,  44,  45,  46,  47,  48,  49,  50]])
```

End of Module

You have reached the end of this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

When you are comfortable with the content, and have practiced to your satisfaction, you may proceed to any related assignments, and to the next module.