Study Unit 5

Introduction to Numpy Array

Introduction to NumPy Array Outline

- Introduction to Numpy
- Installation instruction guide
- How to import Numpy library
- Python Lists and NumPy Arrays
- Array Slicing
- Numpy Attributes
- Other ways to create arrays
- Multi-dimensional array
- Systems of Linear Equations

Study Unit Duration

This Study Unit requires a minimum of 3 hours' formal study time. You may spend an additional 2-3 hours on revision.

Preamble

NumPy is a Python package/library that stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array object. It is also a linear algebra for python and almost all of the libraries in the Python ecosystem rely on it as one of their main building blocks.

Learning Outcomes of Study Unit

Upon completion of this study unit, you should be able to:

5.1 Create one dimensional NumPy Arrays from Python list, access values stored within them and do some mathematical operations

5.2 Create n-dimensional Arrays and carry out different mathematical operations on them

5.3 Work with NumPy Array use cases

Terminologies, Acronyms and their Meaning

AI	Artificial Intelligence
ML	Machine Learning
RL	Reinforcement Learning
DL	Deep learning
EDA	Exploratory Data Analysis
NaN	Not a Number

np	NumPy
pd	Pandas
os	Operating system
AI	Artificial Intelligence
TF	TensorFlow
NULL	Missing value

5.1 Installation Instruction guide

If you installed the Anaconda distribution of Python - it includes Python, NumPy, and other commonly used packages for scientific computing and data science. Therefore, no further installation steps are necessary. We recommend you use the Anaconda distribution of Python as you begin your data science journey.

If you use a version of Python from python.org or a version of Python that came with your operating system, the **Anaconda Prompt** and **conda** or **pip** can be used to install NumPy.

5.1.1 Install NumPy with the Anaconda Prompt

To install NumPy, open the Anaconda Prompt and type:

```
conda install numpy
```

Type **y** for yes when prompted.

5.1.2 Install NumPy with pip

To install NumPy with pip, bring up a terminal window and type:

```
pip install numpy
```

This command installs NumPy in the current working Python environment.

5.1.3 Importing the NumPy module

There are several ways to import NumPy. The standard approach is to use a simple import statement:

```
import numpy
```

However, for large amounts of calls to NumPy functions, it can become tedious to write **numpy.X** over and over again. Instead, it is common to import under the briefer name **np**:

Numpy has many built-in functions and to use each of the functions, we will need to call them from numpy. For example **numpy.array**, **numpy.mean**, **numpy.log10**, etc. Here, we see that we are calling **numpy.something** over and over again. Instead, it is common in Python to use alias. For example:

```
import numpy as np
```

5.1.4 Python Lists and NumPy Arrays

This section introduces NumPy arrays then explains the difference between Python lists and NumPy arrays. NumPy is used to construct homogeneous arrays and perform mathematical operations on arrays. A NumPy array is different from a Python list. The data types stored in a Python list can all be different.

```
python_list = ["Ethiopia", 5, 17.9, True]
type(python_list)
```

```
<class ‘list’>
```

The Python list above contains four different data types:

- “Ethiopia” is a string
- 5 is an integer
- 17.9 is a float
- True is a boolean.

5.2 Working with NumPy Array

NumPy arrays are a bit like Python lists, but still very much different at the same time. The simplest way to create an array in Numpy is to use Python List.

```
myPythonList = [1, 4, 7, 2, 12, 17]
```

We can convert python list to a numpy array by using the object np.array.

```
numpy_array_from_list = np.array(myPythonList)
type(numpy_array_from_list)
```

```
<class ‘numpy.ndarray’>
```

To display the output:

```
numpy_array_from_list
```

```
array([ 1, 4, 7, 2, 12, 17])
```

In practice, there is no need to declare a Python List. The operation can be combined.

```
new_array  = np.array([1, 9, 8, 3, 12])
new_array
```

```
array([ 1, 9, 8, 3, 12])
```

5.2.1 Mathematical operations on one dimensional array

You can perform mathematical operations like additions, subtraction, division, and multiplication on an array. The syntax is the array name followed by the operation (+, -, *, /).

Example 1

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

Addition

```
a + 2
```

```
array([ 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

Subtraction

```
a - 3
```

```
array([-2, -1, 0, 1, 2, 3, 4, 5, 6, 7])
```

Multiplication

```
a * 2
```

```
array([ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20])
```

Division

```
a/2
```

```
array([0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

Exponentiation

```
a**3
```

```
array([ 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000], dtype=int32)
```

When standard mathematical operations are used with arrays, they are applied on an element-by-element basis. This means that the arrays should be the same size during addition, subtraction, etc.

Example 2

```
a = np.array([5, 2, 7, 4, 12])

b = np.array([3, 2, 4, 1, 6])

print(a + b)
```

```
[ 8 4 11 5 18]
```

```
print(a - b)
```

```
[2 0 3 3 6]
```

```
print(a * b)
```

```
[15 4 28 4 72]
```

```
print(b / a)
```

```
[0.6 1. 0.57142857 0.25 0.5]
```

```
print(a % b) # Remainder when you divide each of the elements in a by b
```

```
[2 0 3 0 0]
```

```
b**a
```

```
array([ 243, 4, 16384, 1, -2118184960], dtype=int32)
```

```
np.sqrt(a) # square root of each of the elements in a
```

```
array([2.23606798, 1.41421356, 2.64575131, 2. , 3.46410162])
```

Class activity 1 (Peer to peer review activity)

Peer to Peer Interaction

Visit the LMS, locate forum activity and participate in the discussion

1. Convert the following Python lists to Numpy arrays:

   even_list = [2, 4, 6, 8, 10]

   odd_list = [1, 3, 5, 7, 9]

2 Add the resulting arrays together and name it **even_odd_array**

3. Print the result of **even_odd_array**

5.2.2 One-dimensional Array Indexing

Array elements are accessed, sliced, and manipulated just like lists.

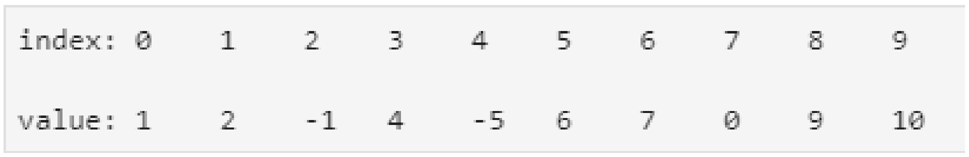
```
a = np.array([1, 2, -1, 4, -5, 6, 7, 0, 9, 10])

print(a)
```

```
[ 1 2 -1 4 -5 6 7 0 9 10]
```

Remember index in Python starts at 0 and ends at n-1.

The index (or location) of each value in the array is shown below:



```
index: 0    1    2    3    4    5    6    7    8    9

value: 1    2    -1   4    -5   6    7    0    9    10
```

The value 1 has an index of 0. We could also say 1 is in location 0 of the array. The value 4 has an index of 3 and the value 8 has an index of 9.

Example 1
Accessing the first element in array a

```
a[0]
```

```
1
```

Example 2
Accessing the third element in array a

```
a[3]
```

```
4
```

Example 3:
Accessing the first element in array a and replacing the result with 5

```
a[2] = 5
a
```

```
array([ 1, 2, 5, 4, -5, 6, 7, 0, 9, 10])
```

Class activity 2 (Peer to peer review activity)

Peer to Peer Interaction

Visit the LMS, locate forum activity and participate in the discussion

Consider:

age_array = np.array([19, 17, 15, 13, 20, 11, 18, 10, 17])

How will you access the 3^{rd} element in the given array?

5.2.3 Array slicing

Values stored within an array can be accessed simultaneously with array slicing. To pull out a section or slice of an array, the colon operator (:) is used when calling the index. The general form is:

array [start : stop]

The index of the slice is specified in **[start : stop]**. Remember Python counting starts at 0 and ends at $n - 1$. The index **[0 : 2]** pulls the first two values out of an array. The index **[1 : 3]** pulls the second and third values out of an array.

Example 1

```
a = np.array([2, 4, 6, 5, 8, 9])
print(a)
```

```
[2 4 6 5 8 9]
```

```
b = a[0:2]
print(b)
```

```
[2 4]
```

Example 2

```
x = np.array([5, 8, 9, 2, 4, 6, 5])
print(x)
```

```
[5 8 9 2 4 6 5]
```

```
print(x[1:3])
```

```
[8 9]
```

On either side of the colon, a blank stand for "default".

- **[:2]** corresponds to **[start=default : stop = 2]**
- **[1:]** corresponds to **[start=1 : stop = default]**

Therefore, the slicing operation **[:2]** pulls out the first and second values in an array. The slicing operation **[1:]** pull out the second through the last values in an array. The examples below illustrate the default stop value is the last value in the array.

Example 3

```
a = np.array([2, 4, 6, 8, 10, 0, 1, 5])
print(a)
```

```
[ 2 4 6 8 10 0 1 5]
```

```
b = a[1:]
print(b)
```

```
[ 4 6 8 10 0 1 5]
```

Example 4

```
x = np.array([0, 5, 6, 1, 0, 8, 1, 5, 9])
print(x)
```

```
[0 5 6 1 0 8 1 5 9]
```

```
y = x[2:]
print(y)
```

```
[6 1 0 8 1 5 9]
```

The next examples show the default **start** value is the first value in the array.

```
a = np.array([2, 4, 6, 8, 1, 3, 6])
print(a)
```

```
[2 4 6 8 1 3 6]
```

```
b = a[:3]
print(b)
```

```
[2 4 6]
```

The following indexing operations output the same array.

```
a = np.array([2, 1, 7, 4, 6, 8, 3])

b = a[0:7] # [start=0:stop= 6]

print(b)
```

```
[2 1 7 4 6 8 3]
```

```
c = a[:7] # [start=0:stop= 6]

print(c)
```

```
[2 1 7 4 6 8 3]
```

```
d = a[0:] # [start=0:stop= 6]

print(d)
```

```
[2 1 7 4 6 8 3]
```

```
e = a[:] # [start=0:stop= 6]

print(e)
```

```
[2 1 7 4 6 8 3]
```

Class activity 3 (Pilot question 1)

Consider:

a = np.array([2, 1, 0, 4, 7, 4, 6, 8, 3])

if you want to pull out the first three values in **a**, that is, to look like:

array([2, 1, 0])

what will you do?

Pilot answer 1

a[:3]

5.2.4 Numpy Attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get the intrinsic properties of the array. Some commonly used attributes are:

- shape: indicates the dimension of an array
- size: returns the total number of elements in the array
- dtype: returns the type of elements in the array, i.e., int64, character

You can check the shape of the array with the object shape preceded by the name of the array. In the same way, you can check the type with dtypes.

Example 1

```
even_array  = np.array([2, 4, 6, 8, 10, 12, 14, 16])

even_array
```

```
array([ 2, 4, 6, 8, 10, 12, 14, 16])
```

dimension of the array

```
print(even_array.shape)
```

```
(8,)
```

total element in the array

```
print(even_array.size)
```

```
8
```

datatype in the array

```
print(even_array.dtype)
```

```
int32
```

Example 2

```
some_array = np.array([12, 62, 65,  7, 21, 60, 10, 87, 14, 43, 51, 10, 38, 95, 26, 11, 46, 47, 34, 68, 58, 77, 13, 10, 45])
some_array
```

```
array([12, 62, 65, 7, 21, 60, 10, 87, 14, 43, 51, 10, 38, 95, 26, 11, 46, 47, 34, 68, 58, 77, 13, 10, 45])
```

```
some_array.shape
```

```
(25,)
```

```
some_array.size
```

```
25
```

5.2.5 Other ways to create arrays

arange

The arange function is similar to the range function but returns an array:

```
import numpy as np
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(1, 10, 2)
```

```
array([1, 3, 5, 7, 9])
```

linspace

Return evenly spaced numbers over a specified interval.

```
np.linspace(0,10,3)
```

```
array([ 0., 5., 10.])
```

```
np.linspace(0, 50, 10)
```

```
array([ 0. , 5.55555556, 11.11111111, 16.66666667, 22.22222222, 27.77777778, 33.33333333,
38.88888889, 44.44444444, 50. ])
```

random

Numpy also has lots of ways to create random number arrays:

rand

numpy.random.rand() creates an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

```
np.random.rand(3)
```

```
array([0.51577564, 0.61414998, 0.59926459])
```

```
np.random.rand(5, 5)
```

```
array([[0.21670278, 0.33876903, 0.63692906, 0.8675732 , 0.82739833], [0.574706 ,
0.17572825, 0.7304413 , 0.80792132, 0.39244966], [0.96046517, 0.40915708, 0.17417772,
0.72472084, 0.48388903], [0.42768233, 0.04920145, 0.34787038, 0.07427598, 0.58396513],
[0.82287729, 0.86726255, 0.32318855, 0.1827099 , 0.22199998]])
```

randn

numpy.random.randn() returns a sample (or samples) from the "**standard normal**" distribution. Unlike **rand** which is **uniform**:

```
np.random.randn(2)
```

```
array([-0.39931173, -1.42514668])
```

```
np.random.randn(50)
```

```
array([ 5.90281962e-01, -2.68820513e+00, 3.55859385e-01, -4.70134624e-01, -6.96299446e-03, 2.26441566e+00, -3.01570019e+00, 5.00142537e-02, 1.57159614e+00, -1.45502043e+00, -1.12834541e+00, -8.09600626e-01, -6.22768305e-01, 4.34774127e-01, -1.03159172e+00, 1.25435863e+00, 6.83860355e-01, 2.17670426e-01, 4.31856702e-01, -2.50393189e-01, -3.83857367e-01, 3.36582061e-01, 6.93761785e-01, 3.41140978e-01, 2.19553456e+00, -2.46751647e+00, -1.34352538e+00, -8.86765471e-01, 1.57982728e+00, 4.91933876e-01, 1.39446883e+00, -2.15174902e+00, -2.20332869e-01, -7.73649981e-01, 7.34320780e-01, -2.51001871e-01, -1.24301411e+00, -6.18636925e-02, -2.30200291e-01, 5.97251915e-01, -1.64933598e+00, -6.01474952e-01, -1.20038663e+00, 7.70133986e-01, 5.10136017e-01, -5.76822347e-01, 9.69298588e-01, 3.99131931e-01, 3.32902425e-01, -1.80469005e-03])
```

```
np.random.randn(5, 5)
```

```
array([[ 0.2331869 , -1.09080222, 1.58275534, 0.27481294, 0.34762717], [ 0.58272839, -0.73934574, -0.32446319, -1.20334318, 0.69339338], [-1.29556187, -0.55346148, -0.01897053, 0.66235769, -0.29137372], [-0.01246607, -1.17154902, 0.00871826, 1.16632516, 1.75843969], [ 1.14889646, 0.44057927, 0.4242706 , 0.22584696, 0.61277228]])
```

randint

Return random integers from low (inclusive) to high (exclusive).

```
np.random.randint(1, 100)
```

```
33
```

```
np.random.randint(1, 100, 10)
```

```
array([90, 12, 73, 58, 20, 17, 89, 39, 6, 69])
```

Class activity 4 (Peer to peer review activity)

Peer to Peer Interaction

Visit the LMS, locate forum activity and participate in the discussion

Create 100 random integers from 20 to 90 using **np.random.randint()** function.

5.3.6 Multi-dimensional array

Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation. A simple 2-D array is defined by a list of lists. Here is an example with a two-dimensional array (e.g. a matrix).

Example 1

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

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A)
```

```
[[1 2 3]
[4 5 6]
[7 8 9]]
```

Dimension of the array:

```
print(A.shape)
```

```
(3, 3)
```

This returns a tuple.

Total elements in the array:

```
print(A.size)
```

```
9
```

Example 2

$$another\_array = \begin{pmatrix} 5 & 6 & 7 \\ 4 & 2 & 1 \\ 3 & 7 & 1 \end{pmatrix}$$

```
another_array = np.array([[5, 6, 7], [4, 2, 1], [3, 7, 1]])

print(another_array)
```

```
[[5 6 7]

[4 2 1]

[3 7 1]]
```

```
print(another_array.shape)
```

```
(3, 3)
```

```
print(another_array.size)
```

```
9
```

Class activity 4 (Peer to peer review activity)

Peer to Peer Interaction

Visit the LMS, locate forum activity and participate in the discussion

Consider:

$$x = \begin{pmatrix} 0 & 6 & 3 \\ 5 & 2 & 7 \\ 3 & 4 & 1 \end{pmatrix}$$

1. Develop a two-dimensional array for x using NumPy
2. What is the shape of x?
3. What is the number of elements in x?

Two-Dimensional (2-D) Array Indexing

Values in a 2-D array can be accessed using the general notation below:

value = array name [row, col]

Where **value** is the value pulled out of the 2-D array and **[row, col]** specifies the row and column index of the value. Remember Python counting starts at 0, so the first row is row zero and the first column is column zero. We can access the value 2 in the array above by calling the row and column index **[0, 1]**. This corresponds to the 1st row (remember row 0 is the first row) and the 2nd column (column 0 is the first column).

$$A = \begin{array}{cccc} C0 & C1 & C2 & \\ 1 & 2 & 3 & R0 \\ 4 & 5 & 6 & R1 \\ 7 & 8 & 9 & R2 \end{array}$$

Example 1

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A)
```

```
[[1 2 3]
 [4 5 6]
[7 8 9]]
```

```
A[0, 1] # value 2 is in row 0 and column 1
```

```
2
```

```
A[1, 0] # value 4 is is row 1 and column 0
```

```
4
```

Example 2

```
a = np.array([[2,3,4],[6,7,8]])
print(a)
```

```
[[2 3 4]

[6 7 8]]
```

```
a[1, 2]
```

```
8
```

```
a[1, 2] = 20

print(a)
```

```
[[ 2 3 4]

[ 6 7 20]]
```

Other array indexing

2D NumPy arrays can also be sliced with the general form:

array[row = start_row : end_row, col = start_col : end_col]

The code section below creates a two row by four column array and indexes out the first two rows and the first three columns.

```
a = np.array([[2, 4, 6, 8], [0, 5, 4, 1]])
print(a)
```

```
[[2 4 6 8]

[0 5 4 1]]
```

```
a.shape
```

```
(2, 4)
```

```
b = a[0:2, 0:3]
print(b)
```

```
[[2 4 6]

[0 5 4]]
```

The code section below slices out the first two rows and all columns from array a.

```
b = a[:2, :]  #[first two rows, all columns]

print(b)
```

```
[[2 4 6 8]

 [0 5 4 1]]
```

Again, a blank represents defaults the first index or the last index. The colon operator all by itself also represents “all” (default start: default stop).

```
b = a[:, :]  #[all rows, all columns]
print(b)
```

```
[[2 4 6 8]
[0 5 4 1]]
```

2D Array mathematics

When standard mathematical operations are used with arrays, they are applied on an element by element basis. This means that the arrays should be the same size during addition, subtraction, etc.

Examples

```
a = np.array([[3, 8], [4, 6]])
print(a)
```

```
[[3 8]
[4 6]]
```

```
b =  np.array([[4, 0], [1, -9]])
print(b)
```

```
[[ 4 0]
[ 1 -9]]
```

Addition

```
a + b
```

```
array([[ 7, 8],
       [ 5, -3]])
```

Subtraction

```
a - b
```

```
array([[-1, 8],
       [ 3, 15]])
```

Multiply by a Constant

```
a*2
```

```
array([[ 6, 16],
       [ 8, 12]])
```

Multiplying by Another Matrix

Multiplication of two matrices is possible only when number of columns in first matrix equals number of rows in second matrix. Multiplication by another matrix uses a **dot product**, np.dot() or with @ operator:

Example

```
np.dot(a, b)
```

```
array([[ 20, -72],
       [22, -54]])
```

```
a @ b
```

```
array([[ 20, -72],
      [ 22, -54]])
```

Class activity 6 (Peer to peer review activity)

Peer to Peer Interaction

Visit the LMS, locate forum activity and participate in the discussion

Consider the following NumPy arrays:

a = np.array([[3, 1, 8], [4,2, 6], [3, 7, 4]])

b = np.array([[4, 0, 6], [1, -9, 2], [1, 2, 3]])

Find the value of:

1. **a - b**
2. **a + b**
3. **a × b**
4. **a[0 : 1, 1 : 2]**

5.3 NumPy use cases: Systems of Linear Equations

Our knowledge in NumPy array can be used to solve system of linear equations. Remember in your junior high school, you were taught how to solve simultaneous equation. The terms simultaneous equations or systems of linear equations refer to conditions where two or more unknown variables are related to each other through an equal number of equations.

A system of linear equations is shown below:

$2x + 3y = 8$

$x - y = -1$

We have two unknowns variables, x and y and two equations i.e. $2x + 3y = 8$ and $x - y = -1$.

Using numpy.linalg.solve()

NumPy's **np.linalg.solve()** function can be used to solve this system of equations for the variables x and y.

Steps to follow:

The steps to solve the system of linear equations with np.linalg.solve() are below:

1. Create NumPy array **A** as a 2 by 2 array of the coefficients in x and y
2. Create a NumPy array **b** as the right-hand side of the equations
3. Solve for the values of x and y using **np.linalg.solve(A, b)**.

The resulting array has two entries. One entry for each variable.

Example 1

Solve simultaneous equation below:

$2x + 3y = 8$

$x - y = -1$

Solution

We want to solve for the unknown x and y.

```
import numpy as np

A = np.array([[2, 3], [1, -1]])

b = np.array([[8], [-1]])

x = np.linalg.solve(A, b)

x
```

```
array([[1.],
       [2.]])
```

x = x[0]

y = x[1]

Therefore, $x = 1$ and $y = 2$.

Example 2

Solve the system of linear equation:

$2x - y = 3$

$x - 3y = -2$

Solution

We want to solve for the unknown x and y.

```
import numpy as np
A = np.array([[2, -1], [1, -3]])
b = np.array([[3], [-2]])
x = np.linalg.solve(A, b)

x
```

```
array([[2.2],

       [1.4]])
```

x = x[0]

y = x[1]

Therefore, $x = 2.2$ and $y = 1.4$.

Class activity 7 (Peer to peer review activity)

Peer to Peer Interaction

Visit the LMS, locate forum activity and participate in the discussion

The following simultaneous/system of equation:

$3x + 4y = 24$

$4x + 3y = 22$

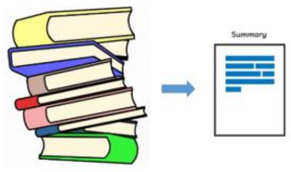
has been broken down to NumPy array for you:

A = np.array([[3, 4], [4, 3]])

b = np.array([[24], [22]])

Complete the following code to solve for x and y:

np.linalg(_ , _)



Summary of Study Unit 5

In this study unit, you have learnt that:

1. NumPy is a Python package/library that stands for Numerical Python.
2. Python Lists are different from NumPy Arrays
3. You can perform mathematical operations on n dimensional arrays
4. .shape, .size, and .dtype are the most commonly used attributes in NumPy Arrays