Lesson # & Title	Focused Area(s)	Lesson Objectives	Graded Tasks (%)
LESSON 2 NumPy and Pandas Data Frame	Topic 1: Introduction to NumPy Topic 2: Using NumPy for basic statistics Topic 3: DataFrame Topic 4: Data ingestion and inspection Topic 5: Exploratory data analysis, Time series in pandas DataFrame	• Understand how to create an array in python for mathematical and logical operations • Understand how to use Pandas library to work with NumPy for data manipulation • How to understand the different parts of the data • Learn about the data structure of panda Series • Learn about the data structure of panda DataFrame	Assignment 20%

Why This Lesson

NumPy and Pandas are the two important libraries used with data analytics. The aim of these libraries is to understand the structure of the data, the operations of the data, and how to explore the data with simple visualisation.

In this lesson, you are required to finish from UNITAR learn online course: DAT208x Introduction to Python Programming.

- Passing grade is 70%
- Finish the virtual lab.

TOPIC 1: Introduction to NumPy

The NumPy (Numerical Python) library is the preferred Python array implementation. As a data analyst, you need to work with numbers, statistics, formulas, and operations on these numbers. Numpy provides the ability to create an array of data and through it, you can do different operations and analyses. Python 3 has a Numpy library no need to do the installation for the library to use it. Here, you will be able to learn how to import it to your solution, and declare different arrays, (i.e. one-dimensional array, and multi-dimensional array).

1.1 Defining Numpy

To use Numpy, first import the library to your solution as follows:

```
# numpy is imported under the shortcut <np>
import numpy as np
```

Now you are able to start working with this library. As mentioned earlier there are two types of arrays here. 1-dimensional array (called Vectors), 2-dimensional array (called matrices). n-dimensional array python can allow additional nested arrays to create different levels or dimensions for the array. To know more about the Numpy and arrays you can write the following command:

```
help( np.ndarray )
```

1.2 Creating your first array

To create an array, you the method np. array, which is used to declare an array with a different number of elements. The argument of the methods can be a list of different numbers as follows:

```
np.array( [1,3,5,7] )
```

Here we defined an array that contains four elements. Apparently, this array contains odd numbers between 0 and 8. To save the data here you assign the above array to a variable. This variable will be of type array.

```
In [4]:
# let's put that array in the variable <arr> so we can work with it
arr = np.array( [1,3,5,7] )
print(arr)
```

```
[1 3 5 7]
```

To access one of the elements, then you can use the same way you access the elements in lists. You put the name of the variable and add to it between two squared brackets the index number.

arr[3] ---------→ output : 7

you can use the slice to return a range of the array elements. The slicer is as follows:

array[start : end] // not the end index will not appear in the output.

```
In [5]:
# we can slice from an array like a list as well
arr[2 :5 ]
```

```
Out[5]: array([5, 7])
```

You may ask why you specified an invalid index "5" where the last index is 3. The answer is simple, Python will check the next index, and if exists, retrieve it. Otherwise, stop. If the end index exists, then it will be a condition to stop and you will not find it in the output. For example, if you change the end to be "3" then the output will be just [5].

```
In [6]:
# we can slice from an array like a list as well
arr[2 :3 ]
```

```
Out[6]: array([5])
```

1.3 2-Dimensional array

To define 2-D array, you can define a list inside each element in the 1-D array's element as follows:

```
In [7]:
# we can similarly define a 2d array
b = np.array( [ [1,2,3],[4,5,6],[7,8,9] ] )
```

This array is of shape 3x3. To find out that, you have the method "shape" which returns the shape of the specified array.

```
In [8]:
# we can similarly define a 2d array
b = np.array( [ [1,2,3],[4,5,6],[7,8,9] ] )
b.shape
```

```
Out[8]: (3, 3)
```

To Do:

1. Try to define an arbitrary array with n-dimensions and show its shape.
2. Use the index and slicer to access the different elements and print the output. (more will be in the coming sections)

1.4 More About Creating Arrays

In the previous section, the array was created using a list. There are more operations that can be done.

a. Array of ones

```
# an array of ones of a specified shape
np.ones( [2, 3] , dtype= np.float64)
```

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

b. Array of zeros

```
# an array of zeroes of a specified shape (note the spelling)
np.zeros( [2, 3], dtype= int )
```

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

c. Eye Array

This method is to create a square matrix with equal indices are 1's otherwise the value is zero.

```
np.eye(4 ,dtype=int)
```

```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

Note that, the elements [0,0], [1,1], [2,2], ….[N, N] has the value 1, and other elements have the value 0.

d. Array Range

np.arrange() is also used to create an array. The range method returns a list of numbers based on the "start", "end", and the "steps".

```
# range equivalent
np.arange( 0, 2.1, 0.2 )
```

```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ])
```

e. Full array

The method used with NumPy is np.full(). The arguments of this method mainly the array shape, and then the value that will fill up all the array elements

```
np.full((3,4), 0.1)
```

```
array([[0.1, 0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1, 0.1]])
```

In this example, the shape of the array is 3x4, and the elements inside the array can be filled with the value of the second argument (0.1).

TOPIC 2: Using NumPy for Basic Statistics

NumPy provides many operators which enable you to write simple expressions that perform operations on entire arrays. First, there is the ability to do elementwise arithmetic operations between arrays. Additionally, there are methods that can be applied to the array like sorting, sum, max, min, mean, median, variance, and standard deviation of the array. There is also the ability to compare arrays using logical operations which is performed element-wise as well.

2.1 Arithmetic Operation on Arrays

The operations on arrays are slightly different from the operations with lists. For example, when you add to lists, this operation will be just a concatenation by merging the two lists in one list with different shapes. However, adding two arrays will result in applying the additional operation elementwise and provide a new array with the same shape.

```
In [9]:
# concatenation
[1,2,3] + [1,2,3]

Out[9]: [1, 2, 3, 1, 2, 3]
```

```
In [10]:
# element-wise addition
np.array( [1,2,3] ) + np.array( [1,2,3] )

Out[10]: array([2, 4, 6])
```

Note: The same fact is applied with the use of variables of array types.

2.2 Operations just for Arrays

There are some operations that are just applicable to arrays not to lists.

a. If you add a number to an array, this operation will add this number to each element in the array. This procedure is called broadcasting. For example, adding "1" to an array

```
# broadcasting
1 + np.array( [1,2,3] )
```

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

b. Product is another example that we should be more careful about. The direct multiplication between two arrays will do an elementwise multiplication of both arrays.

```
# element-wise product
np.array( [1,2,3] ) * np.array( [1,2,3] )
```

```
array([1, 4, 9])
```

[1*1 , 2*2, 3*3]. However, this operation is not the same operation that is used for matrices. The role to do matrix multiplication between two arrays is different and does not follow the elementwise way. To do the matrix multiplication you can use the method "matmul(array1, array2)".

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

```
B = np.array( [[0,1],[1,0]] )
```

```
A * B
```

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

```
np.matmul( A, B )
```

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

You can find the difference between the result of A*B and the np.matmul(A,B).

2.3 Vectorised Functions and Arrays

The array operations are more effective when using a vectorised function to do these operations. The running time to perform these functions is shorter compared to the other operations, with NumPy there are methods that can be used for such operations. They can be unary, where the method will be applied on one array. For example, computing the sine values of each item in the array. The binary method applies the operation on two arrays. For example, for adding two arrays, or using matrix multiplication.

TOPIC 3: DataFrame

A DataFrame is an enhanced two-dimensional array. It can have custom row and column indices and offer additional operations and capabilities that make them more convenient for many data-science-oriented tasks. DataFrames also support missing data. Each column in a DataFrame is a Series. The Series representing each column may contain different element types, as you'll soon see when we discuss loading datasets into DataFrames. Pandas display DataFrames in tabular format with the indices left-aligned in the index column and the remaining columns' values right-aligned. The dictionary's keys become the column names and the values associated with each key become the element values in the corresponding column.

3.1 Define Dataframe

To start using this library, first import the library.

```
# pandas is typically imported as pd
import pandas as pd
```

DataFrames can be defined using different ways. One way is using a dictionary that contains different lists, where the key in the dictionary will represent the column name and the values (lists) as the data.

```
data = {
    "Name": ["Alice", "Bob", "Carol"],
    "Height": [1.45, 1.83, 1.34],
    "Age": [23, 45, 91],
}
```

```
pd.DataFrame( data )
```

	Name	Height	Age
0	Alice	1.45	23
1	Bob	1.83	45
2	Carol	1.34	91

Another method is by reading a dataset/table that contains rows and columns. If the table file does not have column names, then you can define the names of each column. Names are important to be able to access the data in the table. To read the files, you can use the method { read_csv() , read_exel() }

```
# note that the file doesn't have to be a csv, despite the name of the method
df = pd.read_csv( "iris-renamed.tsv", sep="\t" )
```

In this example, we read from a file with extension .tsv, which tab-separated values. The variable df will be of type Dataframe, it will take all the values in the file and treat them as rows and columns.
It is important to understand the shape of the DataFrame, describe it, remove wrong data, and fix any errors before doing the analysis.

```
df.head(10)
```

	sepal_width	sepal_length	petal_width	petal_length	label
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

This method shows the first 10 rows of the dataset. If no argument number is specified as input, then the default will be 5.

It is clear that the data has column names and indexed starts from 0. To print/retrieve the column names you can use the df.col method as follows.

```
df.columns
```

```
Index(['sepal_width', 'sepal_length', 'petal_width', 'petal_length', 'label'], dtype='object')
```

The indexing of the Dataframe can be done using the Dataframe name followed by the table name, then the index in the position of the data.

- df.lable.head()

label
Iris-setosa
Iris-setosa
Iris-setosa
Iris-setosa
Iris-setosa

OR

- df ["label"]. head()

3.2 Indexing Rows

In order to read a specific row, it is important to know which row/index you are looking for. There are two ways:

a. df.iloc

TOPIC 4: Data ingestion and Inspection

When Working with DataFrame, it is necessary to be able to specify the different elements using the indices. Like NumPy, the index of the element will be according to its position in the row and column. Row and column numbers (or names) are used to allocate the element in the DataFrame. In addition, there is a need to understand the data using different methods such as the shape of the Dataframe, that return how many rows and how many columns. Describing the data to understand the statistical values for all the numerical columns. Info of the data, to know the data type of each column.

4.1 The indexing of Dataframe

Indexing can be done using the Dataframe name followed by the table name, then the index in the position of the data.

- df.lable.head()

label
Iris-setosa
Iris-setosa
Iris-setosa
Iris-setosa
Iris-setosa

OR

- df ["label"]. head()

4.2 Indexing Rows

In order to read a specific row, it is important to know which row/index you are looking for. There are two ways:

b. df.iloc

```
# rows are indexed with the .iloc attribute
df.iloc[0]
```

```
sepal_width             5.1
sepal_length            3.5
petal_width             1.4
petal_length            0.2
label           Iris-setosa
Name: 0, dtype: object
```

Here the request is to retrieve the data of row 1. Notify that the name of the columns is available.

c. df.loc()

d. In addition, to specify a range of rows by using the slicer as follows:

```
df.loc[1:5, ["petal_width","label"] ]
```

	petal_width	label
1	1.4	Iris-setosa
2	1.3	Iris-setosa
3	1.5	Iris-setosa
4	1.4	Iris-setosa
5	1.7	Iris-setosa

e. In this case, you define the slicer and also you can specify from which columns.

TOPIC 5: Exploratory data analysis, Time series in pandas DataFrame

To explore data for analysis, there are different methods supported by the library "pandas". You can use the method described, to give you an image and analysis of the data, such as minimum and maximum values, the average, and the median.

```
# descriptive statistics for numerical columns
df.describe( )
```

	sepal_width	sepal_length	petal_width	petal_length
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

To zoom in through these statistics for a specified range, we can use the methods described as follows:

```
# same idea, but only for rows of a certain label
df[df.label == "Iris-setosa"].describe( )
```

	sepal_width	sepal_length	petal_width	petal_length
count	50.00000	50.000000	50.000000	50.00000
mean	5.00600	3.418000	1.464000	0.24400
std	0.35249	0.381024	0.173511	0.10721
min	4.30000	2.300000	1.000000	0.10000
25%	4.80000	3.125000	1.400000	0.20000
50%	5.00000	3.400000	1.500000	0.20000
75%	5.20000	3.675000	1.575000	0.30000
max	5.80000	4.400000	1.900000	0.60000

More methods that can be used for statistics are: min(), max(), mean(), median(), mode()

```
df.std()
```

```
sepal_width     0.828066
sepal_length    0.433594
petal_width     1.764420
petal_length    0.763161
dtype: float64
```

```
df.median()
```

```
sepal_width     5.80
sepal_length    3.00
petal_width     4.35
petal_length    1.30
dtype: float64
```

- end of lesson content –