Lesson # & Title	Focused Area(s)	Lesson Objectives	Graded Tasks (%)
LESSON 4 Manipulating DataFrames with pandas	Topic 1: Handling Missing Data Topic 2: Dropping null values Topic 3: Filling null values Topic 4: Maintaining Not-Null Values	• Understand how to use Python to explore the data and find the null values • Able to reduce the effect of the missing values by dropping the rows/columns containing null values, or filling them with appropriate values • To explore the different data and find the records with different problems such as duplication and outlier data.	**Assignment 2 (30%)**

Why This Lesson

Data is the resource for analytics. These data can be found in different resources as will be explained in the next lessons. Python has libraries that allow the developers to extract the data, visualise the data and transform the data into a new form and bring new information. However, data sometimes are not clean and have many noises that should be considered before the analysis. Most of the time spent is on finding the noise and then solving the existing problems and creating a clean dataset that is ready for processing. In this lesson, different problems will be introduced such as null values, and/or wrong values or data types.

TOPIC 1: Handling Missing Data

1.1 Important functions.

Python has two main libraries that are needed to work with datasets. Pandas and Numpy as discussed in Lesson 2. There are different functions used to test if the numerical data contains null values. Next, these functions are explained.

First, you need to identify the Pandas and Numpy libraries as follows.

```
import numpy as np
import pandas as pd
```

Pandas have different methods that can be used to test if the data contains null values.

a. `pd.isnull(np.nan)`

In this example, the isnull() method tests the content if it contains a null value or not. The method returns true if it contains null, or false otherwise. The content passed to this method is np.nan, which defined a null value, so the output clearly is true. You can define a null value using NumPy or you can use "None" instead.

b. `pd.isna(np.nan)`

In this example, it shows another way to test the content if null or not using the method isna().

c. `pd.notnull(None)`

This method is the opposite of the previous methods where it tests if the content does not contain a null value, which returns true if no null value exists.

Find out the output of the following examples:

- `pd.isnull(pd.Series([1, np.nan, 7]))`

```
df=pd.DataFrame({
    'Column A': [1, np.nan, 7],
    'Column B': [np.nan, 2, 3],
    'Column C': [np.nan, 2, np.nan]
})
```

- `pd.isnull(df)`

1.2 Working with missing values

There are different operations using pandas that manage the null values.

a. `pd.Series([1, 2, np.nan]).count()`

The output of this line is counting the number of accurate values and ignoring null values. In this example, there are just two values and a null value. So, the output will be "2"

b. `pd.Series([1, 2, np.nan]).sum()`

Sum() method will just calculate the numbers while ignoring null values. So the total here will be "3".

c. `pd.Series([2, 2, np.nan]).mean()`

The average method "mean()" will compute the average of just the accurate values and divide by the count of these values. In this example, it will consider just [2,2] values and the sum will be divided by 2.

1.3 Filtering null values

There are other ways to filter null values from the data. From the example below:

```
s = pd.Series([1, 2, 3,None, np.nan, 4])
```

s contains six elements, four numbers and two null values. If we use the isnull() method and call the sum() method, it will return the number of true values as depicted in the line below.

```
pd.isnull(s).sum()
```

The output of this line is "2" as the number "true" values returned by isnull() method is two.

TOPIC 2: Dropping Null Values

After discovering the existence of null values, one way is just to remove that null value. If you have a 2-D dataset, that means the drop would be either the whole row or the whole column. If the number of null rows/columns is more than some acceptable percentage, let's say 10% of the total rows, then it will not be a problem to just drop the rows/columns. Next, we will give an example of how to go through the Dataframe and explore the data, then deal with missing values.

Consider the example below, there is a 4x4 Dataframe defined as below.

```
df = pd.DataFrame({
    'Column A': [1, np.nan, 30, np.nan],
    'Column B': [2, 8, 31, np.nan],
    'Column C': [np.nan, 9, 32, 100],
    'Column D': [5, 8, 34, 110],
})
```

When exploring df, we get this result:

	Column A	Column B	Column C	Column D
0	1.0	2.0	NaN	5
1	NaN	8.0	9.0	8
2	30.0	31.0	32.0	34
3	NaN	NaN	100.0	110

You can use df.info() which will give you the details of the Dataframe columns how many not-null values and the type of each column. After applying the df.isnull() method we get the following result:

	Column A	Column B	Column C	Column D
0	False	False	True	False
1	True	False	False	False
2	False	False	False	False
3	True	True	False	False

It is clear that true means null values and false means numeric values as in the example above.

The method used to drop the null is called "dropna()". calling this function will remove any row containing null values. That means if we applied this on the Dataframe defined above, then the result will be just one row, which is row number 3. (Try this out).

To drop null values based on the columns, we need to specify using the option "axis=1" which will tell Python to drop any column containing null values. The output of this on the Dataframe will be just column D, which is clean and does not contain any null value.

There are other options that can be used to define like a condition of a number of nulls in each row or column based on a threshold, or to drop if all the row is null. (Find out how!)

TOPIC 3: Filling Missing Values

Drop is not the best solution when the percentage of rows or columns that contains null values is huge. Therefore, there are different methods that can be used to fill in the missing value. Here we will explain some of these methods.

3.1 Arbitrary Value

The method used to fill any null value with any specified value is "fillna()". When applied to a series or Dataframe, it will replace the null value with the value specified in the method. For example, consider the series that we defined earlier as follows.

```
s = pd.Series([1, 2, 3,None, np.nan, 4])
```

After that we executed the following line:

```
s2= s.fillna(1)
```

S2 will contain the following output:

[1, 2, 3, 1, 1, 4]

However, using any number could be without meaning or could affect the overall analysis. For that, we can use any statistical methods or any algorithm that can predict the best value that can be filled in this space. For example, we can fill the null values with the average value.

There is an option used with fillna() called "method". This option can take:

- "ffill": which means forward fill, the value is forwarded to the null value. In the previous example, if we use this option, then the result will be [1,2,3,3,3,4]. This means the value "3" is forwarded to the next null.
- "bfill": backward full, is the opposite, the value comes after the null value is backward to replace the null value. That means the output of the previous example will be [1,2,3,4,4,4]

These two options are not always applicable as if the null value is the first element, then you cannot use "ffill" option, or if the null value is the last element, the "bfill" will not be applicable as well. For good practice, mixing these methods explained above would give a better result.

TOPIC 4: Maintaining Not-Null Values

Null values are not always the problem that analyst faces. The data could contain wrong values or wrong data types. These problems can be summarised as follows:

- Inconsistent column names
- Missing data
- Outliers
- Duplicate rows
- Untidy
- Need to process columns
- Column types can signal unexpected data value

Try to find out the problems that can be found in the Figure below.

	Continent	Country	female literacy	fertility	population
0	ASI	Chine	90.5	1.769	1.324655e+09
1	ASI	Inde	50.8	2.682	1.139965e+09
2	NAM	USA	99.0	2.077	3.040600e+08
3	ASI	Indonésie	88.8	2.132	2.273451e+08
4	LAT	Brésil	90.2	1.827	NaN

Figure 1 Problems in data

4.1 Cleaning not-null data

Previously, we studied the cases of dealing with null values. However, there are other issues with data like duplicate rows or data being untidy with different data types in one column for example, or column names and columns that need to be processed or scaled. Another problem is that data contains a wrong value as in the example below.

```
df = pd.DataFrame({
    'Sex': ['M', 'F', 'F', 'D', '?'],
    'Age': [29, 30, 24, 290, 25],
})
df
```

The column "Sex", contains values that are either male "M" or female "F. However you can find that there are other values such as "D" and "?". The letter D could be by mistake pressed from the keyboard as it is next to the "F" letter. What about "?"?

In addition, in the column age, there is an expected value which is 290 and if you visualise this column, you can see the unusual trend of age data as shown in Figure 2. Unfortunately, the system could accept that from the user, but it is totally wrong. As a result, there is a need to process these issues and find ways to solve these problems.

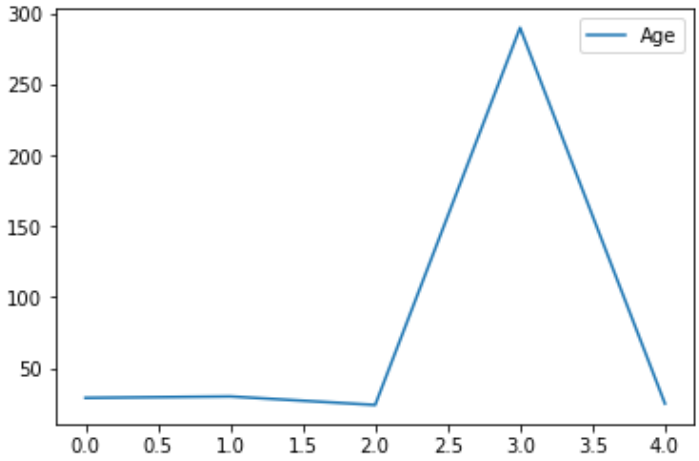

Figure 2 Outlier Data

Now, If you are dealing with a huge amount of data, then the best thing to do is to identify the values in these data. It is easy with categorical data to recognise the incorrect values like Sex in the previous example and needs more work for continuous values. To identify the values, we can use the method "unique()" which returns the unique values in the column.

```
df['Sex'].unique()
```

This line will return: M, F, D,?

It is obvious that there are wrong inputs as explained earlier. The easiest way is to use the function "replace()" to replace the letter D with the letter "F".

```
df['Sex'].replace('D', 'F')
```

For age, we can also replace 290 with 29, as we expect that 0 was added wrongly during the entry. We can do all the replacements in one command as follows:

```
df2=df.replace({
    'Sex': {
        'D': 'F',
        'N': 'M',

    },
    'Age': {
        290: 29
    }
})
```

4.2 Cleaning duplication

There are different ways to discover duplicates in series and Dataframes. Pandas provide a method called "duplicated()" which returns true or false. For example, in the previous example, if we test if the column "Sex" contains duplication then we simply perform the following line:

```
df['Sex'].duplicated()
```

The output is:

```
0    False
1    False
2     True
3    False
4    False
Name: Sex, dtype: bool
```

As you can find in the output, the first two rows are false, but row number three is true which is a duplication of row number two "F". to drop the duplication there is a method called "drop duplicates()" that drops any duplications and keeps the first one occurred in the data. There is an option "keep" that takes arguments like "last" which means drop any duplicate but keep the last one. The default for this is "first" and if you want to drop all duplicates then you put the argument "false".

4.3 Handling text problems

Cleaning text values is not an easy task. It is not like categorical or numerical values where you can spend some time to figure out the problems. With texts, if you have thousands and more records, then it is difficult to correct typos for example. This process may take 90% of the type if not more to clean these data.

- end of lesson content –