Week - 5: NumPy and Pandas - II

Introduction to Pandas:-

Pandas is a library specifically for data analysis; it is built using NumPy. You will be using Pandas extensively for data manipulation, visualisation, building machine learning models, etc.

Pandas is one of the most used libraries in Python, and this is because of the powerful data constructs that it offers. You will learn about the data constructs as you move ahead in the session. But first, to initialise the Pandas library, you can use the following command:
import pandas as pd

In this session, you will learn about:

- Creating dataframes
- Importing CSV data files as Pandas dataframes
- Reading and summarising dataframes
- Sorting dataframes
- Labelling, indexing and slicing data
- Merging dataframes using joins
- Pivoting and grouping

Basics of Pandas:-

Pandas has two main data structures:

- Series
- Dataframes

The more commonly used data structure are DataFrames. So, most of this session will be focused on DataFrames. When you encounter series data structure, Behzad will explain them briefly to you. Let's begin the session by introducing Pandas DataFrames.

DataFrame

It is a table with rows and columns, with rows having an index each and columns having meaningful names. There are various ways of creating dataframes, for instance, creating them from dictionaries, reading from .txt and .csv files. Let’s take a look at them one by one.

Creating dataframes from dictionaries:-

If you have data in the form of lists present in Python, then you can create the dataframe directly through dictionaries. The ‘key’ in the dictionary acts as the column name and the ‘values’ stored are the entries under the column.

Data Frame

In [8]:
```
# All imports
import numpy as np
import pandas as pd
```

Example - 1

Create a Data Frame cars using raw data stored in a dictionary

In [9]:
```
cars_per_cap = [809, 731, 588, 18, 200, 70, 45]
country = ['United States', 'Australia', 'Japan', 'India', 'Russia', 'Morocco', 'Egypt']
drives_right = [True, False, False, False, True, True, True]
```

In [10]:
```
data = {"cars_per_cap": cars_per_cap, "country": country, "drives_right": drives_right}
```

In [11]:
```
data
```

Out[11]:
```
{'cars_per_cap': [809, 731, 588, 18, 200, 70, 45],
 'country': ['United States',
  'Australia',
  'Japan',
  'India',
  'Russia',
  'Morocco',
  'Egypt'],
 'drives_right': [True, False, False, False, True, True, True]}
```

In [12]:
```
cars = pd.DataFrame(data)

cars
```

Out[12]:

	cars_per_cap	country	drives_right
0	809	United States	True
1	731	Australia	False
2	588	Japan	False
3	18	India	False
4	200	Russia	True
5	70	Morocco	True
6	45	Egypt	True

To create a dataframe from a dictionary, you can run the following command:

pd.DataFrame(dictionary_name)

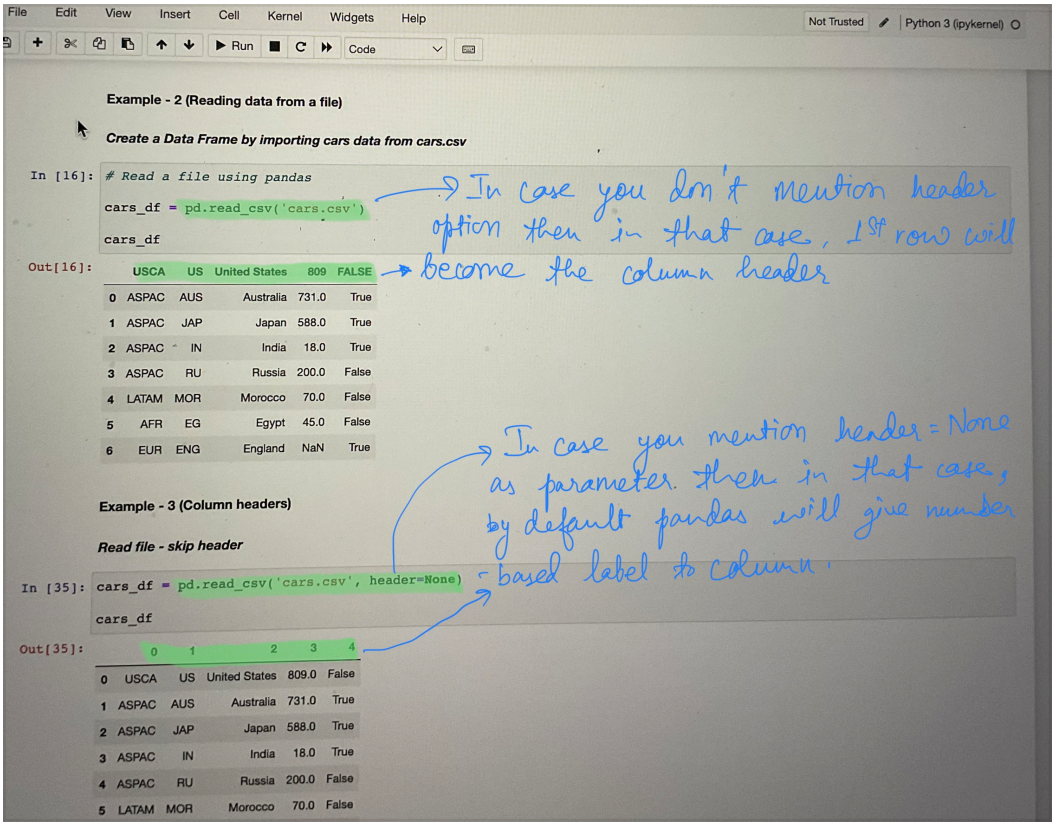
You can also provide lists or arrays to create dataframes, but then you will have to specify the column names as shown below.

pd.DataFrame(list_or_array_name, columns = ['column_1', 'column_2'])

Creating dataframes from external files

Another method to create dataframes is to load data from external files. Data may not necessarily be available in the form of lists. Mostly, you will have to load the data stored in the form of a CSV file, text file, etc. Let's watch the next video and learn how to do that.

Download the file provided 'cars.csv' before you proceed.



Pandas provides the flexibility to load data from various sources and has different commands for each of them. You can go through the list of commands here. The most common files that you will work with are csv files. You can use the following command to load data into a dataframe from a csv file:

pd.read_csv(filepath, sep=',', header='infer')

You can specify the following details:

- separator (by default ‘,’)
- header (takes the top row by default, if not specified)
- names (list of column name)

Pandas - Rows and Columns:-

An important concept in Pandas dataframes is that of the row and column indices. By default, each row is assigned indices starting from 0, which are represented to the left of the dataframe. For columns, the first row in the file (csv, text, etc.) is taken as the column header. If a header is not provided (header = none), then the case is similar to that of row indices (which start from 0).

Pandas library offers the functionality to set the index and column names of a dataframe manually. Let's now learn how to change or manipulate the default

indices and replace them with more logical ones.

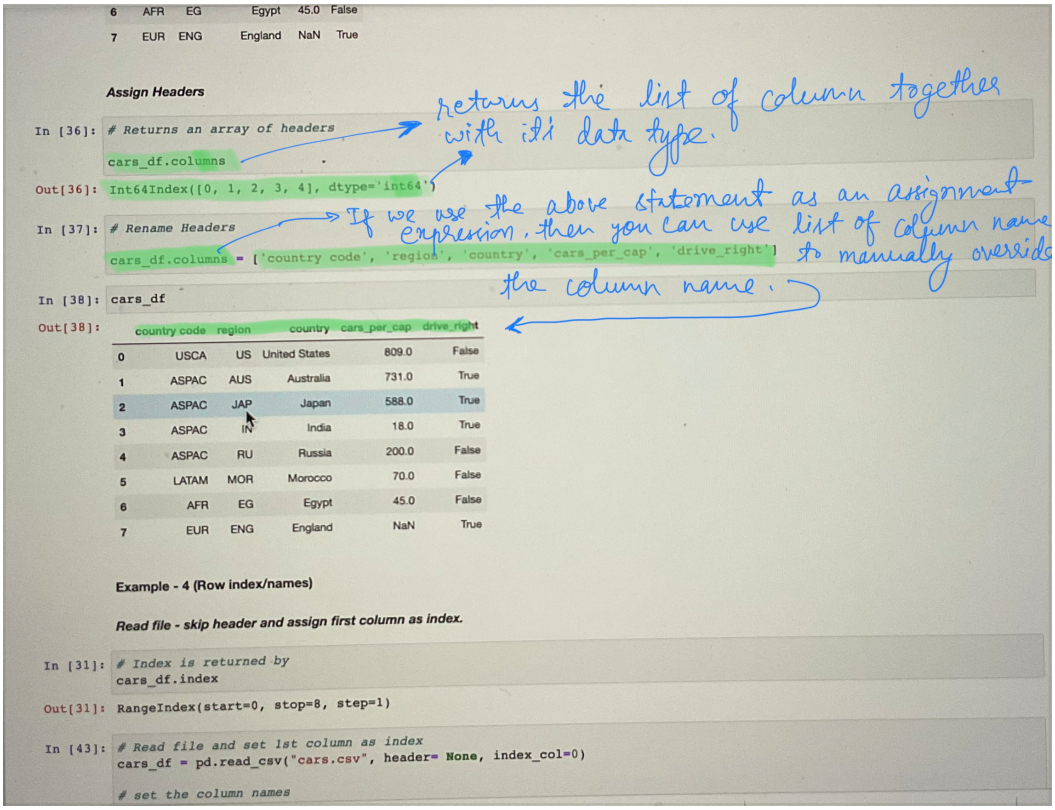
You can use the following code to change the row indices:
dataframe_name.index

To change the index while loading the data from a file, you can use the attribute 'index_col':
pd.read_csv(filepath, index_col = column_number)

It is also possible to create a multilevel indexing for your dataframe; this is known as **hierarchical indexing**.

For column header, you can specify the column names using the following code:

dataframe_name.columns = list_of_column_names



Example - 4 (Row index/names)

As we all know that in Pandas dataframe only rows have index values (i.e. number) while column have labels.

Read file - skip header and assign first column as index.

```
In [31]: # Index is returned by
         cars_df.index
```

this will return the starting, last/stop index value together with step to create the sequence of index present.

```
Out[31]: RangeIndex(start=0, stop=8, step=1)
```

```
In [43]: # Read file and set 1st column as index
         cars_df = pd.read_csv("cars.csv", header= None, index_col=0)

         # set the column names
         cars_df.columns = ['region', 'country', 'cars_per_cap', 'drive_right']
         cars_df
```

This says that use the value present in 0th column as the value to be used as index for rows in dataframe.

Absolute location of the csv to read file data.

This says don't take 1st row as column label.

Out[43]:

0	region	country	cars_per_cap	drive_right
USCA	US	United States	809.0	False
ASPAC	AUS	Australia	731.0	True
ASPAC	JAP	Japan	588.0	True
ASPAC	IN	India	18.0	True
ASPAC	RU	Russia	200.0	False
LATAM	MOR	Morocco	70.0	False
AFR	EG	Egypt	45.0	False
EUR	ENG	England	NaN	True

index as

index name is '0' by default.

```
In [44]: # Print the new index
         cars_df.index
```

```
Out[44]: Index(['USCA', 'ASPAC', 'ASPAC', 'ASPAC', 'ASPAC', 'LATAM', 'AFR', 'EUR'], dtype='object', name=0)
```

Rename the Index Name

```
In [46]: cars_df.index.name = 'country_code'
         cars_df
```

way to override the default value of row index name.

Out[46]:

country_code	region	country	cars_per_cap	drive_right

index name is "country_code" changed by above code.

USCA	US	United States	809.0	False
ASPAC	AUS	Australia	731.0	True
ASPAC	JAP	Japan	588.0	True
ASPAC	IN	India	18.0	True
ASPAC	RU	Russia	200.0	False
LATAM	MOR	Morocco	70.0	False
AFR	EG	Egypt	45.0	False
EUR	ENG	England	NaN	True

Delete the index name

If you make the row index name as "None" then it will not have any name (custom name) or not even default name '0'.

```
In [51]: cars_df.index.name = None
         cars_df
```

Out[51]:

	region	country	cars_per_cap	drive_right
USCA	US	United States	809.0	False
ASPAC	AUS	Australia	731.0	True
ASPAC	JAP	Japan	588.0	True
ASPAC	IN	India	18.0	True
ASPAC	RU	Russia	200.0	False
LATAM	MOR	Morocco	70.0	False
AFR	EG	Egypt	45.0	False
EUR	ENG	England	NaN	True

that's why no name for row index.

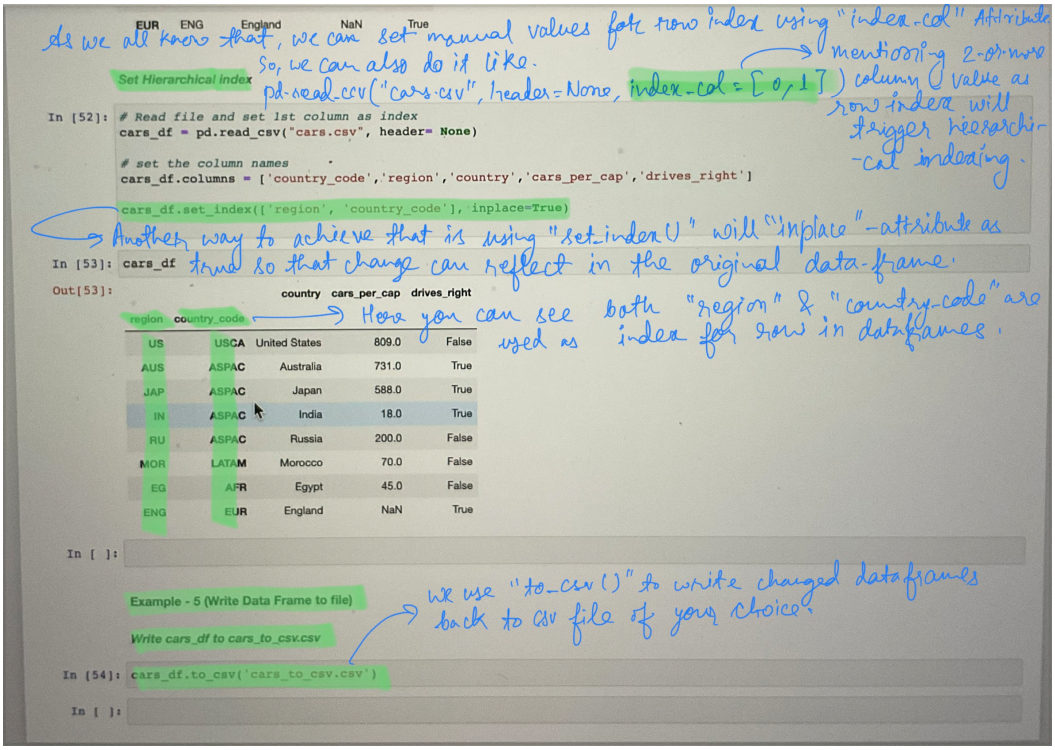
Set Hierarchical index

```
In [52]: # Read file and set 1st column as index
         cars_df = pd.read_csv("cars.csv", header= None)

         # set the column names
         cars_df.columns = ['country_code','region','country','cars_per_cap','drives_right']

         cars_df.set_index(['region', 'country_code'], inplace=True)
```

```
In [53]: cars_df
```



Describing Data:-

In the previous segment, you learnt how to load data into a dataframe and manipulate the indices and headers to represent the data in a meaningful manner. Let's first load the data that will be used in the demonstrations in this segment. You can use the Jupyter Notebook provided below to code along with the instructor.

Behzad explains the use of index_col to create a hierarchically indexed DataFrame. The information is only partially correct, only index_col will create a multi-index DataFrame, not sort it. If you want to sort it you will have to use the sort_index() method. Example code:
rating = pd.read_csv("rating.csv", header = 0, index_col = [2,1])
rating.sort_index()

Now that you know how hierarchical indexing is done and you understand its benefits, let's learn the ways of extracting information from a DataFrame. In this segment, you will learn some basic functions that will be useful for describing the data stored in the dataframes. The same notebook will be used in the next segment as well.

While working with Pandas, the dataframes may hold large volumes of data; moreover, it would be an inefficient approach to load the entire data whenever an operation is performed. Hence, you must use the following code to load a limited number of entries:
dataframe_name.head()

By default, it loads the first five rows, although you can specify a number if you want fewer or more rows to be displayed. Similarly, to display the last entries, you can use the tail() command instead of head().

In the video, you learnt about two commands which give statistical information as well:

- dataframe.info(): This method prints information about the dataframe, which includes the index data type and column data types, the count of non-null values and the memory used.
- dataframe.describe(): This function produces descriptive statistics for the dataframe, that is, the central tendency (mean, median, min, max, etc.), dispersion, etc. It analyses the data and generates output for both numeric and non-numeric data types accordingly.

Indexing and Slicing:-

There are multiple ways to select rows and columns from a dataframe or series. In this segment, you will learn how to:

- Select rows from a dataframe
- Select columns from a dataframe
- Select subsets of dataframes

Selection of rows in dataframes is similar to the indexing that you saw in NumPy arrays. The syntax df[start_index:end_index] will subset the rows according to the start and end indices.

However, you can have all the columns for each row using the function provided above. With the introduction of column labels, selecting columns is no more similar to that in arrays. Let's watch the next video and learn how to select the required column(s) from a dataframe.

You can select one or more columns from a dataframe using the following commands:

- df['column'] or df.column: It returns a series
- df[['col_x', 'col_y']]: It returns a dataframe

Pandas series data type:

To visualise pandas series easily, it can be thought of as a one-dimensional (1D) NumPy array with a label and an index attached to it. Also, unlike NumPy arrays, they can contain non-numeric data (characters, dates, time, booleans, etc.). Usually, you will work with Series only as part of dataframes.

You could create a Pandas series from an array-like object using the following command:
pd.Series(data, dtype)

The methods taught above allow you to extract columns. But how would you extract a specific column from a specific row?

You can use the loc method to extract rows and columns from a dataframe based on the following labels:

dataframe.loc[[list_of_row_labels], [list_of_column_labels]]

This is called **label-based indexing** over dataframes. Now, you may face some challenges while dealing with the labels. As a solution, you might want to fetch data based on the row or column number.

As you learnt in the video, another method for indexing a dataframe is the iloc method, which uses the row or column number instead of labels.

dataframe.iloc[rows, columns]

Since positions are used instead of labels to extract values from the dataframe, the process is called **position-based indexing**. With these two methods, you can easily extract the required entries from a dataframe based on their labels or positions. The same set of commands, loc and iloc , can be used to slice the data as well.

Subsetting rows based on conditions

Often, you want to select rows that meet some given conditions. For example, you may want to select all orders where Sales > 3,000, or all orders where 2,000 < Sales < 3,000 and Profit < 100. Arguably, the best way to perform these operations is to use df.loc[], since df.iloc[] would require you to remember the integer column indices, which is tedious. Let's start first with one condition to filter the elements in the dataframe.

As you can see, you can easily segregate the entries based on the single or multiple conditions provided. To get the desired results by subsetting the data, it is important to have well-written conditional statements.

You already know the basic conditional operators like "<" or ">". There are a couple of other functions which might come in really handy while handling real-life datasets. These are isin() and isna().

- isin() : Similar to the membership operator in lists, this function can check if the given element "is in" the collection of elements provided.
- isna() : It checks whether the given element is null/empty.

Tranformations

You can use pandas to transform columns into more readable forms. For instance, a sales value of 34876$ per month is difficult to read. It can be simply rounded to 35k, that way it becomes easy to read. Transformation is not a very widely used technique, you can listen to the video linked here to learn more about it.

In the next segment, you will learn how to run operations over the dataframes; this will help you create or modify the stored data.

Display Markets with Sales >300000

```
In [28]: sales["Sales"] > 300000
```

```
Out[28]: Region
         Western Africa       False
         Southern Africa      False
         North Africa         False
         Eastern Africa       False
         Central Africa       False
         Western Asia         False
         Southern Asia         True
         Southeastern Asia     True
         Oceania               True
         Eastern Asia          True
         Central Asia         False
         Western Europe        True
         Southern Europe      False
         Northern Europe      False
         Eastern Europe       False
         South America        False
         Central America       True
         Caribbean            False
         Western US           False
         Southern US          False
         Eastern US           False
         Central US           False
         Canada               False
         Name: Sales, dtype: bool
```

```
In [29]: sales[ sales["Sales"] > 300000 ]
```

Out[29]:

Region	Market	No_of_Orders	Profit	Sales
Southern Asia	Asia Pacific	469	67,998.76	351,806.60
Southeastern Asia	Asia Pacific	533	20,948.84	329,751.38
Oceania	Asia Pacific	646	54,734.02	408,002.98
Eastern Asia	Asia Pacific	414	72,805.10	315,390.77
Western Europe	Europe	964	82,091.27	656,637.14
Central America	LATAM	930	74,679.54	461,670.28

Display the LATAM and Eruopean countries with sales > 250000

```
In [30]: sales[  (sales["Market"].isin(["LATAM", "Europe"])) & (sales["Sales"] > 250000)     ]
```

Out[30]:

Region	Market	No_of_Orders	Profit	Sales
Western Europe	Europe	964	82,091.27	656,637.14
Northern Europe	Europe	367	43,237.44	252,969.09
Central America	LATAM	930	74,679.54	461,670.28

Display Sales and Profit data for Western Africa Southern Africa and North Africa

```
In [23]: sales.loc[["Western Africa", "Southern Africa", "North Africa"] , ["Sales", "Profit"]]
```

Out[23]:

Region	Sales	Profit
Western Africa	78,476.06	-12,901.51
Southern Africa	51,319.50	11,768.58
North Africa	86,698.89	21,643.08

```
In [24]: sales.iloc[0:3, 2:4]
```

Out[24]:

Region	Profit	Sales
Western Africa	-12,901.51	78,476.06
Southern Africa	11,768.58	51,319.50
North Africa	21,643.08	86,698.89

```
In [24]: sales.iloc[0:3, 2:4]
```

Out[24]:

Region	Profit	Sales
Western Africa	-12,901.51	78,476.06
Southern Africa	11,768.58	51,319.50
North Africa	21,643.08	86,698.89

Example - 5 (Transformation)

Replace the sales values in the form of thousands

Context: Some time you might want to modify columns to make them more readable. For instance, the sales column in the given data set has six digits, followed by two decimal places. You might want to make it more readable. You can convert the actual sales number to a number in thousands and make it a round figure.

eg. 300000 - 300K

You can use the .floordiv function to achieve the transformation explained above. You can read more about the .floordiv method here.

```
In [18]: sales.Sales = sales.Sales.floordiv(1000)

         sales.head()
```

Out[18]:

Region	Market	No_of_Orders	Profit	Sales
Western Africa	Africa	251	-12,901.51	78.00
Southern Africa	Africa	85	11,768.58	51.00
North Africa	Africa	182	21,643.08	86.00
Eastern Africa	Africa	110	8,013.04	44.00
Central Africa	Africa	103	15,606.30	61.00

```
In [19]: sales.rename(columns={'Sales': 'Sales in Thousands'}, inplace=True)
         sales.head()
```

Out[19]:

Region	Market	No_of_Orders	Profit	Sales in Thousands
Western Africa	Africa	251	-12,901.51	78.00
Southern Africa	Africa	85	11,768.58	51.00
North Africa	Africa	182	21,643.08	86.00
Eastern Africa	Africa	110	8,013.04	44.00
Central Africa	Africa	103	15,606.30	61.00

Replace values in Profit percent of total

```
In [20]: sales.head()
```

Out[20]:

Operations on Dataframes:-

So far, in this session, you have been working with a dummy data set, and the functions that were performed on the data were more theoretical than practical. From this point, you will be working with a real-life data set. Find the dataset and the notebook used from this point onwards linked below.

The data set contains weather data from Australia, and the same data is being used by an FMCG company to predict sales in their stores. Before going into the details of the tasks that you will be performing in this session.

So, as you saw in the video, the data set contains weather data from Australia. Take a look at the data dictionary below:

1. **Date**: Date on which a data was recorded
2. **Location**: The location where the data was recorded
3. **MinTemp**: Minimum temperature on the day of recording data (in degrees Celsius)
4. **MaxTemp**: Maximum temperature on the day of recording data (in degrees Celsius)
5. **Rainfall**: Rainfall in mm
6. **Evaporation**: The so-called Class A pan evaporation (mm) in the 24 hours up to 9 AM
7. **Sunshine**: Number of hours of bright sunshine in the day
8. **WindGustDir**: Direction of the strongest gust of wind in the 24 hours up to midnight
9. **WindGustSpeed**: Speed (km/h) of the strongest gust of wind in the 24 hours up to midnight

The type of data that is recorded after a specific time period is called time-series data. The data being used in this case study is an example of Time-series data. The observations in the data are recorded periodically after 1 day. We will have a brief discussion on how to work with time-series data a bit later in the session; for now, let's focus on the data set and the tasks associated with it.

You are already familiar with the filter function, which was used to solve the problem given in the video. Similar to the filtering in NumPy, running a Pandas DataFrame through a conditional statement also returns boolean values. These boolean values can be used to slice out data.

The next important task that you will learn is to create new columns in the DataFrame. Frankly speaking, creating new columns is as simple as assigning a column to the output of an operation that you are carrying out. Although it might seem not so simple but it really is, it will become clear after you watch the demonstration.

To create new columns, you will use the time-series functionality. So, before moving on to the actual demonstration, let's discuss a time series briefly next.

Handling time-series data

Time-series data refers to a series of data points that are indexed over time. The data is recorded over regular time intervals and is stored along with the time it was recorded. Some common examples of time series include stock prices, temperature, weather report, etc., as this information would make sense

only when presented with the time it was recorded.

If a date–time variable has values in the form of a string, then you can call the 'parse_dates' function while loading the data into the Pandas DataFrame. This will convert the format to date–time for that particular variable. Fortunately, no such data-type conversion is required in the given data set. In the upcoming video, you will learn how to extract the series data.

once data is loaded in a date–time format, Pandas can easily interpret the different representations of date and time.

Apart from handling time-series data, another important feature of a DataFrame is the user-defined functions. In the previous module, you have already seen that lambda functions are the most accessible of all types of user-defined functions. Let's take a look at lambda functions once again before proceeding further.

Lambda functions

Suppose you want to create a new column 'is_raining', which categorises days into rainy or not rainy days based on the amount of rainfall. You need to implement a function, which returns 'Rainy' if rainfall > 50 mm, and 'Not raining' otherwise. This can be done easily by using the apply() method on a column of the DataFrame.

you saw the use of the 'apply()' method to apply a simple lambda function on a column in the DataFrame. Now, the next step is to add the data in a new column to the DataFrame.

786 rows × 13 columns

```
[27]: asw_df["Rainfall"].apply(lambda x: "Rainy" if x > 50 else "Not Rainy")
```

```
[27]: 0          Not Rainy
      1          Not Rainy
      2          Not Rainy
      3          Not Rainy
      4          Not Rainy
                   ...
      142188     Not Rainy
      142189     Not Rainy
      142190     Not Rainy
      142191     Not Rainy
      142192     Not Rainy
      Name: Rainfall, Length: 142193, dtype: object
```

```
[28]: asw_df["is_raining"] = asw_df["Rainfall"].apply(lambda x: "Rainy" if x > 50 else "Not Rainy")
```

```
[29]: asw_df.head(20)
```

[29]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	Year	Month	Dayofmonth	Maxtemp_F	is_raining
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	2008	12	1	73.22	Not Rainy
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	2008	12	2	77.18	Not Rainy

The columns that are created by the user are known as '**Derived Variables**'. Derived variables increase the information conveyed by a DataFrame. Now, you

can use the lambda function to modify the DataFrames.

In the upcoming segments, you will learn how to use the groupby function to aggregate the created DataFrame.

Groupby and Aggregate Functions:-

Grouping and aggregation are two of the most frequently used operations in data analysis, especially while performing exploratory data analysis (EDA), where it is common to compare summary statistics across groups of data.

As an example, in the weather time-series data that you are working with, you may want to compare the average rainfall of various regions or compare temperature across different locations.

A grouping analysis can be thought of as having the following three parts:

- **Splitting the data** into groups (e.g., groups of location, year, and month)
- **Applying a function** on each group (e.g., mean, max, and min)
- **Combining the results** into a data structure showing summary statistics

the groupby function is quite a powerful function, and it can significantly reduce the work of a data scientist. The groupby() function returns a Pandas object, which can be used further to perform the desired aggregate functions.

you will see another example, which is a high-level problem, wherein you will not only use the groupby and the aggregate function but also use a user-defined function to create a new column; you can then apply the groupby and the aggregate function over this column. It is a bit complex to reiterate the problem, and it is alright if you feel lost; you can watch the video again to get a better understanding. The learning from this video will not affect your journey further in this session.

That was a fun example, was it not? Now, before moving any further, note that if you apply the groupby function on an index, you will not encounter any error while executing the grouping and aggregation commands together. However, when grouping on columns, you should first store the DataFrame and then run an aggregate function on the new DataFrame.

Merging DataFrames:-

In this segment, you will learn how to merge and concatenate multiple DataFrames. In a real-world scenario, you would rarely have the entire data stored in a single table to load into a DataFrame. You will have to load the data into Python using multiple DataFrames and then find a way to bring everything together.

This is why merge and append are two of the most common operations that are performed in data analysis. You will now learn how to perform these tasks using different DataFrames. First, let's start with merging. The data set that you have been working with contains only weather data and no sales data. Sales data is stored in a different data set. How would you combine these data sets? Let's find out...

You can use the following command to merge the two DataFrames above:

```
dataframe_1.merge(dataframe_2, on = ['column_1', 'column_2'], how = '_____')
```

In the next video, we will take a look at the useful attribute 'how', which is provided by the merge function.

The attribute how n the code above specifies the type of merge that is to be performed. Merges are of the following different types:

- left: This will select the entries only in the first dataframe.
- right: This will consider the entries only in the second dataframe.
- outer: This takes the union of all the entries in the dataframes.
- inner: This will result in the intersection of the keys from both frames.

Depending on the situation, you can use an appropriate method to merge the two DataFrames.

Concatenating dataframes

Concatenation is much more straightforward than merging. It is used when you have dataframes with the same columns and want to stack them on top of each other, or with the same rows and want to append them side by side.
You can add columns or rows from one dataframe to another using the concat function:
`pd.concat([dataframe_1, dataframe_2], axis = _)`

To append rows, you have to set the axis value as 0. For adding columns from one dataframe to another, the axis value must be set as 1. If there are any extra columns or rows where there are no values, they are replaced with 'NaN'.

You can also perform various mathematical operations between two or more dataframes. For example, you may have two dataframes for storing the sales information for 2018 and 2019. Now, you want the sales data combined for a period of two years. In such a case, the add function in Pandas allows you to directly combine the two dataframes easily.

Apart from the merge, append or concat, you can perform mathematical operations to combine multiple dataframes. When two dataframes have the

same row and column labels, you can directly use the mathematical operators provided in the list below:

- add(): +
- sub(): -
- mul(): *
- div(): /
- floordiv(): //
- mod(): %
- pow() :**

Pandas will return the derived values with the same labels in a combined dataframe. It also provides the attribute fill_value to control how you want to deal with the values that are not common between two dataframes. You can refer to the documentation for the same. For a better understanding of these, function explores the following notebook.

Pivot Tables:-

A pivot table is quite a useful tool to represent a DataFrame in a structured and simplified manner. It acts as an alternative to the groupby() function in Pandas. Pivot tables provide excel-like functionalities to create aggregate tables.

you can use the following command to create pivot tables in Pandas:

```
df.pivot(columns='grouping_variable_col', values='value_to_aggregate', index='grouping_variable_row')
```

The pivot_table() function can be used to also specify the aggregate function that you would want Pandas to execute over the columns that are provided. It could be the same or different for each column in the DataFrame. You can write the pivot_table command as shown below:

```
df.pivot_table(values, index, aggfunc={'value_1': np.mean,'value_2': [min, max, np.mean]})
```

The function above, when substituted with proper values, will result in a mean value of value_1 and three values (minimum, maximum and a mean of value_2) for each row.

Summary:-

Let's summarise what you have learnt in this session. You learnt about the Pandas library, which provides various functions to conduct data analysis in Python.

The various topics that were covered are:

- Pandas Series and Dataframes, which are the basic data structures in the Pandas library
- Indexing, selecting and subsetting a dataframe
- Merging and appending two dataframes, which can be done using the .merge and .concat commands
- Grouping and summarising dataframes, which can be done using groupby() to first make an object and then use it to play around
- The pivot table function in a dataframe, which is similar to pivot tables in MS Excel
- Finally, you learnt how to perform different functions over time-series data using Pandas