

## Facilitation Guide

### Index

- I. Recap
- II. Read and save data in a csv file.
- III. Cleaning Data.

(2 hrs) ILT

5mins Recap of Previous Session

In this session we are going to understand how to read data from a .csv file and write data into a .csv file. Also we are going to understand different ways to clean our data.

### I. Read and save data in a csv file.

#### Read Data from csv file:

#### What is a CSV?

CSV stands for “Comma Separated Values”. It is the simplest form of storing data in tabular form as plain text. It is important to know how to work with CSV because we mostly rely on CSV data in our day-to-day lives as data scientists.

#### Structure of CSV in Python



YearsExperience	Salary
1.1	39343.00
1.3	46205.00
1.5	37731.00
2.0	43525.00
2.2	39891.00
2.9	56642.00
3.0	60150.00
3.2	54445.00
3.2	64445.00

## How to Read CSV Files in Python Using Pandas?

To read CSV files in Python using Pandas, you can use the `pd.read_csv()` function provided by the Pandas library.

### **Syntax:**

```
pd.read_csv(filepath, sep=',', header='infer', names=None, usecols=None, encoding='utf-8', engine='c')
```

Here are the most commonly used parameters for `pd.read_csv()`:

**filepath:** This is the path to the CSV file you want to read. It can be a file path, URL, or a buffer-like object.

**sep (default: ','):** The delimiter used to separate values in the CSV file. By default, it's a comma, but you can specify other characters like tab ('\t') or a semicolon(';').

**header (default: 'infer'):** This parameter specifies which row in the CSV file should be considered as the column names (header). Use 'infer' to automatically detect the header row based on the presence of column names.

**names:** A list of column names to use if you want to specify column names explicitly instead of using the header row in the CSV.

**usecols:** A list of columns to read from the CSV file. You can select specific columns by providing a list of column names or column indices.

**encoding (default: 'utf-8'):** The character encoding to use when reading the CSV file. Common encodings include 'utf-8', 'iso-8859-1', 'cp1252', etc.

**engine (default: 'c'):** The parsing engine to use. You can choose 'c' (C engine), 'python' (Python engine), or 'python-fwf' (Python engine for fixed-width formatted files).

### **Here are the steps to read a CSV file:**

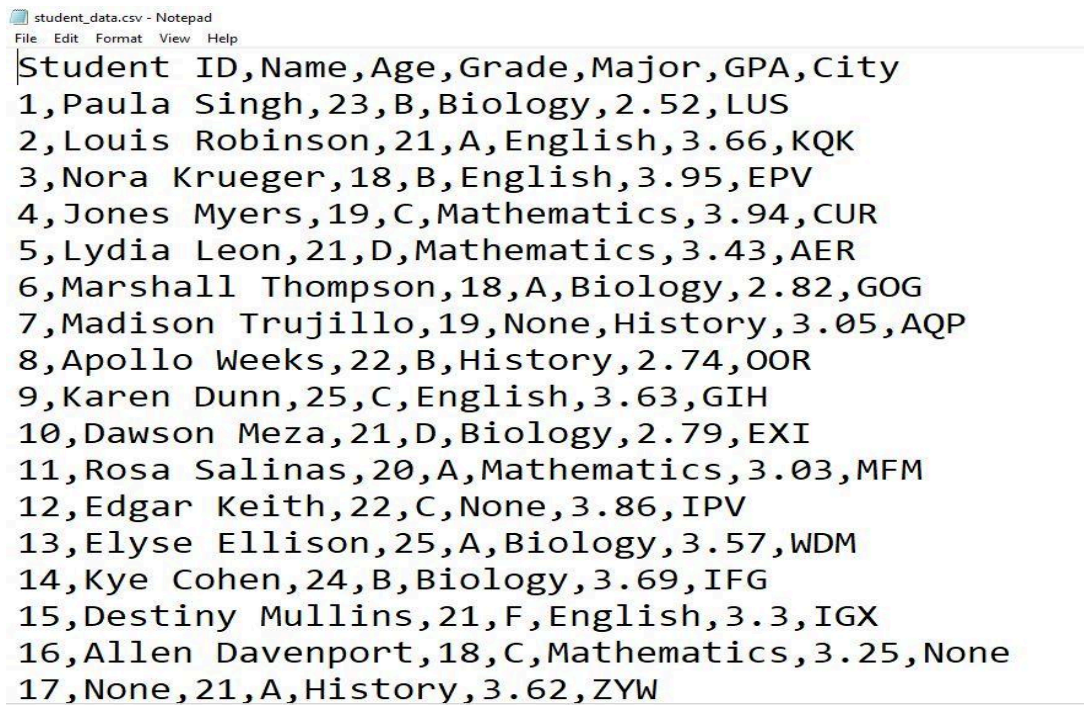
Import the Pandas library in your Python script or Jupyter Notebook:

```
import pandas as pd
```

Use the `pd.read_csv()` function to read the CSV file into a Pandas DataFrame.

Provide the path to the CSV file as an argument to this function.

For example, if you have a CSV file named `student_data.csv`:



```
Student ID,Name,Age,Grade,Major,GPA,City
1,Paula Singh,23,B,Biology,2.52,LUS
2,Louis Robinson,21,A,English,3.66,KQK
3,Nora Krueger,18,B,English,3.95,EPV
4,Jones Myers,19,C,Mathematics,3.94,CUR
5,Lydia Leon,21,D,Mathematics,3.43,AER
6,Marshall Thompson,18,A,Biology,2.82,GOG
7,Madison Trujillo,19,None,History,3.05,AQP
8,Apollo Weeks,22,B,History,2.74,00R
9,Karen Dunn,25,C,English,3.63,GIH
10,Dawson Meza,21,D,Biology,2.79,EXI
11,Rosa Salinas,20,A,Mathematics,3.03,MFM
12,Edgar Keith,22,C,None,3.86,IPV
13,Elyse Ellison,25,A,Biology,3.57,WDM
14,Kye Cohen,24,B,Biology,3.69,IFG
15,Destiny Mullins,21,F,English,3.3,IGX
16,Allen Davenport,18,C,Mathematics,3.25,None
17,None,21,A,History,3.62,ZYW
```

```
# Provide the path to your CSV file
csv_file_path = 'student_data.csv'

# Use read_csv to read the data from the CSV file into a DataFrame
data_df = pd.read_csv(csv_file_path)
```

You can now work with the data stored in the Pandas DataFrame (`data_df`).

For example, you can display the first 5 rows by default of the DataFrame using the `.head()` method:

**Syntax:**

**`data_frame.head()`:**

`dataframe.head(n)`

**dataframe** is the Pandas DataFrame you want to display the first rows of.  
series is a Pandas Series.

**n** is an optional parameter that specifies the number of rows to display. If you don't provide **n**, it defaults to 5, which means it will show the first 5 rows by default.

Eg.

```
data_frame.head() # this will display the first 5 rows by default
```

```
data_frame.head(7) # displays the first seven rows
```

```
data_frame.head(n=5) # here n is parameter with value 5
```

To display the data, use the below line of code.

```
print(data_df.head())
```

**Here's a complete example:**

```
import pandas as pd

# Provide the path to your CSV file
csv_file_path = 'student_data.csv' # Replace with the actual path to your CSV file

# Use read_csv to read the data from the CSV file into a DataFrame
data_df = pd.read_csv(csv_file_path)

# Display the first few rows of the DataFrame
print(data_df.head())
```

**Output:**

	Student ID	Name	Age	Grade	Major	GPA	City
0	1	Paula Singh	23	B	Biology	2.52	LUS
1	2	Louis Robinson	21	A	English	3.66	KQK
2	3	Nora Krueger	18	B	English	3.95	EPV
3	4	Jones Myers	19	C	Mathematics	3.94	CUR
4	5	Lydia Leon	21	D	Mathematics	3.43	AER

**Note:** Download the student\_data.csv file from your LMS

## How to write CSV Files in Python Using Pandas?

You can write CSV files in Python using Pandas by using the `to_csv()` method of a Pandas DataFrame.

### **to\_csv() function**

The `to_csv()` function in Pandas is used to write the data from a Pandas DataFrame to a CSV (Comma-Separated Values) file. It allows you to save your DataFrame as a structured text file that can be easily shared and used by other applications.

### **Syntax:**

```
DataFrame.to_csv(path_or_buf, sep=',', columns=None, header=True, index=True, mode='w')
```

### **Here are some of the commonly used parameters:**

**path\_or\_buf:** This is the file path or a file-like object (e.g., a file handle or a string buffer) where you want to save the CSV data.

**sep (default: ','):** The delimiter to use to separate values in the CSV file.

**columns:** A list of column names or labels to include in the CSV file. If not provided, all columns are included.

**header (default: True):** Determines whether to write the column names as the first row in the CSV file.

**index:** If True, the DataFrame index will be written to the file. If False, no index will be written.

**mode (default: 'w'):** The file mode for writing the CSV file. Common modes include 'w' (write), 'a' (append - it will open the existing file and then write the data below to the file). Create a Pandas DataFrame or use an existing one.

For example, let's create a simple DataFrame:

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
}
```

```
df = pd.DataFrame(data)
```

Specify the file path where you want to save the CSV file. Replace 'output.csv' with the desired file path and name:

```
csv_file_path = 'output.csv'
```

Use the `to_csv()` method to write the DataFrame to a CSV file. Specify the file path in the `path_or_buf` parameter:

```
df.to_csv(csv_file_path, index=False)
```

The `index` parameter is set to `False` to prevent writing the DataFrame's index to the CSV file. You can set it to `True` if you want to include the index in the CSV file.

**Here's the complete code:**

```
import pandas as pd

# Create a Pandas DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
}

df = pd.DataFrame(data)

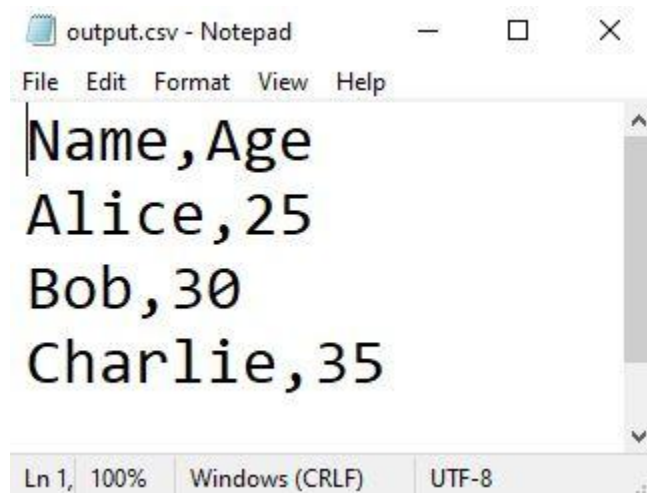
# Specify the file path to save the CSV file
csv_file_path = 'output.csv'

# Write the DataFrame to a CSV file
df.to_csv(csv_file_path, index=False) # this will create a new file in the current folder

print(f'Data has been written to {csv_file_path}')
```

After running this code, you'll have a CSV file named output.csv in the specified location with the data from the DataFrame. You can customize the data, column names, and the CSV file path as needed.

### Output:



## II. Cleaning Data.

Our data often comes from multiple resources and is not clean. It may contain missing values, duplicates, wrong or undesired formats, etc. Running your experiments on this messy data leads to incorrect results. Therefore, it is necessary to prepare your data before it is fed to your model. This preparation of the data by identifying and resolving the potential errors, inaccuracies, and inconsistencies is termed as Data Cleaning.

### Steps for Data Cleaning

#### 1. Loading the Dataset

Load the Iris dataset using Pandas' read\_csv() function:

#### Example:

```
import pandas as pd
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']
data = pd.read_csv('student_data.csv', names= column_names, header=0)
print(data.head())
```

### Explanation:

We import the Pandas library using the `import pandas as pd` statement, making Pandas available for use in our script.

We define a list `column_names` that contains the column names we want to assign to the DataFrame. These column names are `['Student ID', 'Name', 'Age', 'Grade', 'Major', 'GPA', 'City']`.

We use the `pd.read_csv()` function to read the data from the `'student_data.csv'` file into a Pandas DataFrame.

`'student_data.csv'` is the file path to the CSV file we want to read.

`names=column_names` specifies that the `column_names` list we defined should be used as the column names of the DataFrame.

`header=0` indicates that the first row of the CSV file (index 0) contains the header with column names.

We store the resulting DataFrame in the variable named `data`.

Finally, we use `data.head()` to display the first 5 rows of the DataFrame to check the data and verify that the column names have been correctly assigned.

### Output:

	Student ID	Name	Age	Grade	Major	GPA	City
0	1	Paula Singh	23	B	Biology	2.52	LUS
1	2	Louis Robinson	21	A	English	3.66	KQK
2	3	Nora Krueger	18	B	English	3.95	EPV
3	4	Jones Myers	19	C	Mathematics	3.94	CUR
4	5	Lydia Leon	21	D	Mathematics	3.43	AER

## 2. Explore the dataset

To get insights about our dataset, we will print some basic information using the built-in functions in pandas.

### info() function



The `info()` function in Pandas is used to display a concise summary of the metadata and information about a DataFrame, including the data types, non-null counts, and memory usage.

### Syntax:

```
DataFrame.info(verbose=True, buf=None, max_cols=None, memory_usage=True, null_counts=None)
```

**Here are the commonly used parameters of the `info()` function:**

#### **verbose (default: True)**

If set to `True`, it provides a detailed summary of the DataFrame. If set to `False`, it provides a shorter summary without the data types of each column.

**buf (default: None):** This parameter is an optional output buffer where the summary is written. If not specified, the summary is printed to the console.

**max\_cols (default: None):** This parameter specifies the maximum number of columns to be displayed in the summary. If there are more columns, some may be truncated.

**memory\_usage (default: True):** If set to `True`, it includes memory usage information in the summary, showing the memory consumed by the DataFrame.

**null\_counts (default: None):** This parameter is used to specify whether to count the number of null (missing) values in the summary. If not provided, the behavior is determined by the global option `'display.null_counts'` in the Pandas options.

#### **describe() function**

The `describe()` function in Pandas is used to generate various summary statistics for numerical columns in a DataFrame. It provides statistics such as count, mean, standard deviation, minimum, maximum, and quartiles.

### Syntax:

```
DataFrame.describe(percentiles=None, include=None, exclude=None)
```

**Here are the commonly used parameters of the `describe()` function:**

**percentiles (default: [.25, .5, .75]):** This parameter allows you to specify which percentiles to include in the summary statistics. By default, it includes the 25th (Q1), 50th (median), and 75th (Q3) percentiles.

**include (default: None):** A list of data types to include in the summary. You can pass data type names (e.g., 'number', 'object') or specify None to include all data types.

**exclude (default: None):** A list of data types to exclude from the summary. You can pass data type names (e.g., 'number', 'object') or specify None to exclude no data types.

### Example:

```
import pandas as pd
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']
data = pd.read_csv('student_data.csv', names= column_names, header=0)
print("Dataframe information")
print(data.info())
print("Dataframe Description")
print(data.describe())
```

### Explanation:

We import the Pandas library using the import pandas as pd statement, making Pandas available for use in our script.

We define a list column\_names that contains the column names you want to assign to the DataFrame. These column names are ['Student ID', 'Name', 'Age', 'Grade', 'Major', 'GPA', 'City'].

We use the pd.read\_csv() function to read the data from the 'student\_data.csv' file into a Pandas DataFrame.

'student\_data.csv' is the file path to the CSV file we want to read.

names=column\_names specifies that the column\_names list we defined should be used as the column names of the DataFrame.

header=0 indicates that the first row of the CSV file (index 0) contains the header with column names.

We store the resulting DataFrame in the variable data.

We print the "Dataframe information" using `data.info()`. This function provides an overview of the DataFrame, including the data types, non-null counts, and memory usage.

We print the "Dataframe Description" using `data.describe()`. This function generates summary statistics for numerical columns in the DataFrame, including count, mean, standard deviation, minimum, maximum, and quartiles.

### Output:

```
Dataframe information
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0   Student ID  100 non-null    int64
 1   Name        100 non-null    object
 2   Age         100 non-null    int64
 3   Grade       100 non-null    object
 4   Major       100 non-null    object
 5   GPA         100 non-null    float64
 6   City        100 non-null    object
dtypes: float64(1), int64(2), object(4)
memory usage: 5.6+ KB
None
Dataframe Description
```

	Student ID	Age	GPA
count	100.000000	100.000000	100.000000
mean	50.500000	21.080000	3.26340
std	29.011492	2.316738	0.43019
min	1.000000	18.000000	2.51000
25%	25.750000	19.000000	2.90750
50%	50.500000	21.000000	3.29500
75%	75.250000	23.000000	3.64500
max	100.000000	25.000000	3.99000

From the above output we can see that first the `info()` function generates the data types, non-null and record count.

Second output shows all the numerical columns statistics.

### 3. Checking Value Distribution

This is an important step in understanding how the data is distributed in categorical columns, which is an important task for classification.

You can perform this step using the `value_counts()` function in pandas.

### **value\_counts() function**

The `value_counts()` function in Pandas is used to count the occurrences of unique values in a Pandas Series or DataFrame column.

It returns a Series containing the counts of unique values, sorted in descending order by default. Here's the syntax for the `value_counts()` function:

#### **Syntax:**

```
DataFrame['column_name'].value_counts(normalize=False, sort=True, ascending=False, dropna=True)
```

#### **Here are some of the commonly used parameters:**

**normalize (default: False):** If set to True, it returns the relative frequencies (proportions) of each unique value rather than the counts.

**sort (default: True):** If set to False, it doesn't sort the result by counts, and the order will be based on the order of appearance in the original data.

**ascending (default: False):** If set to True, it sorts the result in ascending order.

**dropna (default: True):** If set to False, it includes counts of NaN or missing values.

#### **Example:**

```
#import pandas
import pandas as pd

#list the columns name
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']

#read the data from the csv file
data = pd.read_csv('student_data.csv', names= column_names, header=0)

#printing the value distribution
print(data['Major'].value_counts())
```

#### **Explanation:**

We import the Pandas library using the `import pandas as pd` statement, making Pandas available for use in our script.

We define a list `column_names` that contains the column names you want to assign to the DataFrame. These column names are `['Student ID', 'Name', 'Age', 'Grade', 'Major', 'GPA', 'City']`.

We use the `pd.read_csv()` function to read the data from the `'student_data.csv'` file into a Pandas DataFrame.

`'student_data.csv'` is the file path to the CSV file we want to read.

`names=column_names` specifies that the `column_names` list we defined should be used as the column names of the DataFrame.

`header=0` indicates that the first row of the CSV file (index 0) contains the header with column names.

We store the resulting DataFrame in the variable `data`.

We use `data['Major'].value_counts()` to count the occurrences of unique values in the `'Major'` column of the DataFrame. This provides a Pandas Series that shows the counts of each unique major.

Finally, we print the result of the `value_counts()` function, which displays the distribution of majors in your dataset.

#### Output:

```
Major
English      25
Biology      24
History      20
Mathematics   18
Computer Science  13
Name: count, dtype: int64
```

From the above output we can see that most of the students' have majored in English.

## 4. Removing Missing Values

The `dropna()` function in Pandas is used to remove missing or NaN (Not-a-Number) values from a DataFrame or Series. It allows you to clean our data by eliminating rows or columns that contain missing values.

### Syntax:

*DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)*

### Here are some of the commonly used parameters:

**axis (default: 0):** Specifies whether you want to drop rows with missing values (axis=0) or columns with missing values (axis=1).

**how (default: 'any'):** Determines when to drop a row or column. It can take the following values:

**'any':** Drops the row or column if it contains any missing values (default).

**'all':** Drops the row or column only if all values are missing.

**thresh (default: None):** Allows you to specify a threshold for the number of non-missing values required to keep a row or column.

**subset (default: None):** A list of columns or index labels to consider when dropping missing values. It is useful when you want to drop missing values only from specific columns.

**inplace (default: False):** If set to True, the DataFrame or Series is modified in place, and no new object is returned. If False, a new DataFrame or Series with missing values removed is returned.

```
#import pandas
import pandas as pd
#list the columns name
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']

#read the data from the csv file
data = pd.read_csv('student_data.csv', names= column_names, header=0)
```

```
#printing the nan values
print("Before Removing nan Values")
print(data.info())

#removing the nan values
data.dropna(inplace=True)
print("After Removing nan Values")
print(data.info())
```

**Explanation:**

We import the Pandas library using the `import pandas as pd` statement, making Pandas available for use in our script.

We define a list `column_names` that contains the column names you want to assign to the DataFrame. These column names are `['Student ID', 'Name', 'Age', 'Grade', 'Major', 'GPA', 'City']`.

We use the `pd.read_csv()` function to read the data from the `'student_data.csv'` file into a Pandas DataFrame.

`'student_data.csv'` is the file path to the CSV file we want to read.

`names=column_names` specifies that the `column_names` list we defined should be used as the column names of the DataFrame.

`header=0` indicates that the first row of the CSV file (index 0) contains the header with column names.

We store the resulting DataFrame in the variable `data`.

We print the information about the DataFrame using `data.info()`. This provides an overview of the DataFrame, including data types, non-null counts, and memory usage.

We use `data.dropna(inplace=True)` to remove rows with missing values (NaN) from the DataFrame. The `inplace=True` parameter means that the DataFrame is modified in place, and no new object is returned.

Finally, we print the information about the DataFrame again using `data.info()` to show that the rows with missing values have been removed.

## Output:

```
Before Removing nan Values
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Student ID  100 non-null   int64
1   Name        99 non-null    object
2   Age         100 non-null   int64
3   Grade       99 non-null    object
4   Major       99 non-null    object
5   GPA         100 non-null   float64
6   City        99 non-null    object
dtypes: float64(1), int64(2), object(4)
memory usage: 5.6+ KB
None

After Removing nan Values
<class 'pandas.core.frame.DataFrame'>
Index: 96 entries, 0 to 99
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Student ID  96 non-null     int64
1   Name        96 non-null     object
2   Age         96 non-null     int64
3   Grade       96 non-null     object
4   Major       96 non-null     object
5   GPA         96 non-null     float64
6   City        96 non-null     object
dtypes: float64(1), int64(2), object(4)
memory usage: 6.0+ KB
None
```

Here from the above first output we can see that in Name, Grade, Major and City column we have one missing value.

In the second output we can see that it removes all the nan values from the above 4 columns and after removing 4 missing values it displays 96 non null values.

**Note:** The non-null count of all other columns also becomes 96 because, the `dropna()` function will delete all the records that have a null value in any of the columns.

## 5. Removing Duplicates

Duplicates can distort our analysis so we remove them from our dataset. We will first check their existence using the below-mentioned command:

### **duplicated() function**



The `deduplicated()` function in Pandas is used to identify and mark duplicated rows in a `DataFrame` or `Series`. It returns a `Boolean Series` that indicates whether each row is a duplicate of a previous row (or multiple previous rows).

### Syntax:

```
DataFrame.duplicated(subset=None, keep='first')
```

### Here are the commonly used parameters:

**subset (DataFrame only, default: None):** A list of column names or labels. If provided, the function checks for duplicates only within the specified subset of columns.

**keep (default: 'first'):** Determines which duplicated values to mark. It can take the following values:

- 'first': Marks all duplicates as True except for the first occurrence.
- 'last': Marks all duplicates as True except for the last occurrence.
- False: Marks all duplicates as True.

### Example:

```
#import pandas
import pandas as pd

#list the columns name
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']

#read the data from the csv file
data = pd.read_csv('student_data.csv', names= column_names, header=0)

#Find the duplicate values
duplicate_rows = data.duplicated()

#Printing the no of duplicate rows
print("Number of duplicate rows:", duplicate_rows.sum())
```

### Explanation:

We import the Pandas library using the `import pandas as pd` statement, making Pandas available for use in our script.

We define a list `column_names` that contains the column names you want to assign to the DataFrame. These column names are `['Student ID', 'Name', 'Age', 'Grade', 'Major', 'GPA', 'City']`.

We use the `pd.read_csv()` function to read the data from the `'student_data.csv'` file into a Pandas DataFrame.

`'student_data.csv'` is the file path to the CSV file we want to read.

`names=column_names` specifies that the `column_names` list we defined should be used as the column names of the DataFrame.

`header=0` indicates that the first row of the CSV file (index 0) contains the header with column names.

We store the resulting DataFrame in the variable `data`.

We use the `data.duplicated()` function to check for duplicated rows in the DataFrame `data`. It returns a Boolean Series that indicates which rows are duplicates.

We calculate the number of duplicate rows by using `duplicate_rows.sum()`. The `.sum()` function is used to count the number of True values in the Boolean Series, which corresponds to the number of duplicate rows.

Finally, we print the number of duplicate rows found in our dataset.

### Output:

```
Number of duplicate rows: 6
   Student ID  Name  Age  Grade  Major  GPA  City
100         99 Vivian Aguirre   19     C  History  2.98  TFB
101        100  Andy Espinoza   18     F  Biology  3.56  WFJ
102         99 Vivian Aguirre   19     C  History  2.98  TFB
103        100  Andy Espinoza   18     F  Biology  3.56  WFJ
104         99 Vivian Aguirre   19     C  History  2.98  TFB
105        100  Andy Espinoza   18     F  Biology  3.56  WFJ
|
```

From the output we can see that in my dataset 6 duplicate records are present

The duplicates can be removed via the **`drop_duplicates()`** function.

### **drop\_duplicates() Function**

The `drop_duplicates()` function in Pandas is used to remove duplicate rows from a DataFrame. It allows you to eliminate rows that have the same values in specified columns, leaving only one unique occurrence of each set of values.

#### **Syntax:**

*DataFrame.drop\_duplicates(subset=None, keep='first', inplace=False)*

#### **Here are the commonly used parameters:**

**subset (default: None):** A list of column names or labels. If provided, the function checks for duplicates only within the specified subset of columns.

**keep (default: 'first'):** Determines which duplicated values to keep. It can take the following values:

- 'first': Keeps the first occurrence of duplicate values (default).
- 'last': Keeps the last occurrence of duplicate values.
- False: Drops all occurrences of duplicate values.

### **inplace (default: False)**

If set to True, the DataFrame is modified in place, and no new object is returned.

#### **Example:**

```
#import pandas
import pandas as pd

#list the columns name
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']

#read the data from the csv file
data = pd.read_csv('student_data.csv', names= column_names, header=0)

#Find the duplicate values
duplicate_rows = data.duplicated()

#Printing the no of duplicate rows
```

```
print("Before Removing Number of duplicate rows:", duplicate_rows.sum())
print(data[duplicate_rows])

#Removing duplicate values
data.drop_duplicates(inplace=True)

#Find the duplicate values
duplicate_rows = data.duplicated()

#Printing the no of duplicate rows
print("After Removing Number of duplicate rows:", duplicate_rows.sum())
print(data[duplicate_rows])
data.to_csv('after_drop.csv', index=False)
```

**Explanation:**

We import the Pandas library using the import pandas as pd statement, making Pandas available for use in our script.

We define a list column\_names that contains the column names you want to assign to the DataFrame. These column names are ['Student ID', 'Name', 'Age', 'Grade', 'Major', 'GPA', 'City'].

We use the pd.read\_csv() function to read the data from the 'student\_data.csv' file into a Pandas DataFrame.

'student\_data.csv' is the file path to the CSV file we want to read.

names=column\_names specifies that the column\_names list we defined should be used as the column names of the DataFrame.

header=0 indicates that the first row of the CSV file (index 0) contains the header with column names.

We store the resulting DataFrame in the variable data.

We use the data.duplicated() function to check for duplicate rows in the DataFrame data. It returns a Boolean Series that indicates which rows are duplicates.

We calculate and print the number of duplicate rows before removing duplicates using duplicate\_rows.sum().

We use `data.drop_duplicates(inplace=True)` to remove duplicate rows from the DataFrame. The `inplace=True` parameter means that the DataFrame is modified in place, and no new object is returned.

We again use the `data.duplicated()` function to check for duplicated rows after removing duplicates.

We calculate and print the number of duplicate rows after removing duplicates using `duplicate_rows.sum()`.

### Output:

```
Before Removing Number of duplicate rows: 6
   Student ID   Name  Age Grade  Major  GPA City
100         99 Vivian Aguirre   19    C  History  2.98  TFB
101        100  Andy Espinoza   18    F  Biology  3.56  WFJ
102         99 Vivian Aguirre   19    C  History  2.98  TFB
103        100  Andy Espinoza   18    F  Biology  3.56  WFJ
104         99 Vivian Aguirre   19    C  History  2.98  TFB
105        100  Andy Espinoza   18    F  Biology  3.56  WFJ
After Removing Number of duplicate rows: 0
Empty DataFrame
Columns: [Student ID, Name, Age, Grade, Major, GPA, City]
Index: []
```

Here from the above output we can see that we successfully deleted all the duplicate values.

Also, since we did not specify the 'keep' parameter, it will retain the first occurrence by default. So you will now have one entry of the record which had duplicates earlier.

Open the 'after\_drop.csv' file and check that only one record is present after removing duplicates.

### Output:

```
after_drop.csv - Notepad
File Edit Format View Help
94,Marley Austin,23,B,Computer Science,3.85,IUZ
95,Alivia Holland,23,A,Biology,3.66,SDG
96,Brady Hubbard,18,B,Biology,2.51,TVV
97,Rosie Chan,24,A,Biology,3.68,HLO
98,Frank Powell,23,C,History,3.24,GPT
99,Vivian Aguirre,19,C,History,2.98,TFB
100,Andy Espinoza,18,F,Biology,3.56,WFJ
```

## 6. Save the cleaned dataset

Save the cleaned dataset to the new CSV file.

```
data.to_csv('<name-of-the-csv', index=False)
```

```
#import pandas
import pandas as pd

#list the columns name
column_names = ['Student ID','Name','Age','Grade','Major','GPA','City']

#read the data from the csv file
data = pd.read_csv('student_data.csv', names= column_names, header=0)

#Before removing checking no of row and col
print("Before removing checking no of row and col")
print(data.shape)

#Remove duplicate values
data.drop_duplicates(inplace=True)

#After removing checking no of row and col
print("After removing checking no of row and col")
print(data.shape)
data.dropna(inplace=True)
print("After removing Nan values no of row and col")
print(data.shape)
#Save the files
data.to_csv('cleaned_data.csv', index=False)
print("Data Saved ")
```

**Output:**

```
Before removing checking no of row and col  
(106, 7)  
After removing checking no of row and col  
(100, 7)  
After removing Nan values no of row and col  
(96, 7)  
Data Saved
```

From the above output we can see that before dropping duplicate values there are 106 rows and 7 columns and after removing the duplicates it's showing 100 rows and 7 columns .

Then we removed all the nan values and after that it showed 96 rows and 7 columns. Then we save the clean data inside the cleaned\_data.csv file.

### Exercise

#### Use GPT to write a program:

1.Hi! I have a dataset containing information about students ,and I want to perform data cleaning such as drop duplicate values,display shape,display info and describe the stats .I want to use dummy data. Can you please generate a complete code for me?

***Please refer to the ILT7 Lab doc in the LMS.***