Pandas

-Pandas is a Python library used for working with data sets.
-It has functions for analyzing, cleaning, exploring, and manipulating data.

***What Can Pandas Do?

Pandas gives you answers about the data.
Like:

- -Is there a correlation between two or more columns?
- -What is average value?
- -Max value?
- -Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

***Where is the Pandas Codebase?

The source code for Pandas is located at this github repository

https://github.com/pandas-dev/pandas

***Installation of Pandas

If you have Python and PIP already installed on a system, then installation of Pandas is very easy.
Install it using this command:

```
C:\Users\Your Name>pip install pandas
```

***Import pandas

Once pandas is installed, import it in your applications by adding the import keyword:

```
import pandas
```

***Checking NumPy Version

```
import pandas as pd
print(pd.__version__)
```

A Pandas series is like a column in a table . it is 1D array which holds data of any type.

- A one-dimensional labeled array capable of holding any data type.

here we will create a simple pandas series.

```
import pandas as pd


arun = [1,7,2]      # list
arunnew = pd.Series(arun)
print(arunnew)
```

```
0    1
1    7
2    2
dtype: int64
```

```python
import pandas as pd

# Creating a Series
data = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(data)
# Output:
# a    10
# b    20
# c    30
# d    40
# dtype: int64

# Accessing data
print(data['b'])  # Output: 20
```

labeling - label can be use to access a specified

```
import pandas as pd


arun = [1,7,2]
arunnew = pd.Series(arun)
print(arunnew[0])
```

```
1
```

with Create label(its an argument) you can create your own name labels:

```
import pandas as pd

arun = [1,7,2]
arunnew = pd.Series(arun, index=["x", "y", "z"])
print(arunnew)
```

```
x    1
y    7
z    2
dtype: int64
```

labeling - label can be use to access a specified value.(after creating own label)

```
arun = [1,7,2]
arunnew = pd.Series(arun, index=["x", "y", "z"])
print(arunnew["x"])
```

```
1
```

you can also use a key or value object like a dictionary, when creating a series.
here we will create a simple pandas series from a dictionary.

```
cal = {"day1": 420, "day2":380, "day3":390}
arun = pd.Series(cal)
print(arun)
```

```
day1    420
day2    380
day3    390
dtype: int64
```

now we will create a series using only data from day1 and day2

```
cal = {"day1": 420, "day2":380, "day3":390}
result = pd.Series(cal, index=["day1", "day2"])
print(result)
```

```
day1    420
day2    380
dtype: int64
```

DataFrame: Data sets in pandas are usually multidimentional tables, and they re called DataFrames.

-it is a 2D data structure like a 2D array with table incl. rows and columns.

series are like columns and dataframes is the whole table.

we will now create a dataframe from 2 series.

```
import pandas as pd
arun = {"cal": [420, 380, 390], "duration": [50, 40, 45]}
arunnew = pd.DataFrame(arun)
print(arunnew)
```

```
   cal  duration
0  420        50
1  380        40
2  390        45
```

Locate row: pandas use the loc attibute to return one or more specified row.

```
import pandas as pd
data = {"cal": [420, 380, 390], "dur":[50, 40, 45]}
arun = pd.DataFrame(data)
print(arun.loc[0])
```

```
cal    420
dur     50
Name: 0, dtype: int64
```

loc (Label-based indexing)

- select data based on row/column labels.
- Includes both the start and end points of a range.

```
import pandas as pd

data = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['x', 'y', 'z'])

# Access by label
print(data.loc['x'])  # Row with label 'x'

# Access multiple rows/columns by labels
print(data.loc[['x', 'z'], ['A']])  # Rows 'x', 'z' and column 'A'

# Conditional selection
print(data.loc[data['A'] > 1])  # Rows where column 'A' > 1
```

iloc (Integer-based indexing)

- select data by row/column positions.
- Similar to Python slicing (end point is excluded).

```
# Access by position
print(arun.iloc[0])  # First row

# Access multiple rows/columns by positions
print(arun.iloc[0:2, 1])  # Rows 0 to 1 (exclusive of 2) and column at position 1
```

```python
# named Index: with the index arg, you can name your own index.
import pandas as pd
data = {"cal": [420, 380, 390], "dur":[50, 40, 45]}
arun = pd.DataFrame(data, index=["day1", "day2", "day3"])
print(arun)

print("============================")

# locate the named index:
import pandas as pd
data = {"cal": [420, 380, 390], "dur":[50, 40, 45]}
arun = pd.DataFrame(data, index=["day1", "day2", "day3"])
print(arun.loc["day2"])

print("============================")

# output in a dataframe:
import pandas as pd
data = {"cal": [420, 380, 390], "dur":[50, 40, 45]}
arun = pd.DataFrame(data, index=["day1", "day2", "day3"])
print(arun.loc[["day1", "day2"]])

print("============================")
```

File I/O

1. Reading Data

```
# Read from CSV
df = pd.read_csv('file.csv')

# Read from Excel
df = pd.read_excel('file.xlsx')

# Read from JSON
df = pd.read_json('file.json')
```

2. Writing Data

```
# Write to CSV
df.to_csv('output.csv', index=False)

# Write to Excel
df.to_excel('output.xlsx', index=False)

# Write to JSON
df.to_json('output.json')
```

#load the data from the csv file into dataframe i.e data.csv

```
import pandas as pd
fileload = pd.read_csv('data.csv')
print(fileload)
```

" " can also be used

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
..	...	...	...	...
164	60	105	140	290.8
165	60	110	145	300.0
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

only top 5 and below 5 data is visible if we directly print fileload

read csv files: (comma seperated file) it is a simple way to store the big and bigest data sets. csv files contains plain text.

loading the csv into a dataframe with to_string

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.to_string())
```

to print the whole data

```
    Duration  Pulse  Maxpulse  Calories
0         60    110       130     409.1
1         60    117       145     479.0
2         60    103       135     340.0
3         45    109       175     282.4
4         45    117       148     406.0
5         60    102       127     300.0
6         60    110       136     374.0
7         45    104       134     253.3
8         30    109       133     195.1
9         60     98       124     269.0
10        60    103       147     329.3
11        60    100       120     250.7
12        60    106       128     345.3
13        60    104       132     379.3
14        60     98       123     275.0
15        60     98       120     215.2
16        60    100       120     300.0
17        45     90       112       NaN
18        60    103       123     323.0
19        45     97       125     243.0
20        60    108       131     364.2
21        45    100       119     282.0
22        60    130       101     300.0
23        45    105       132     246.0
24        60    102       126     334.5
25        60    100       120     250.0
26        60     92       118     241.0
27        60    103       132       NaN
28        60    100       132     280.0
29        60    102       129     380.3
30        60     92       115     243.0
31        45     90       112     180.1
32        60    101       124     299.0
33        60     93       113     223.0
34        60    107       136     361.0
35        60    114       140     415.0
36        60    102       127     300.0

[169 rows x 4 columns
```

```
# max_rows : you can check your system's maximum rows with:
import pandas as pd
print(pd.options.display.max_rows)
```

```
60
```

```python
# yes, we can increase the maximum number of rows to display the entire dataframe.
import pandas as pd
pd.options.display.max_rows = 169
df = pd.read_csv('data.csv')
print(df)
```

now print the whole data, if <=168, then it will still print the same top and below 5 rows

JSON File: (JavaScript Object Notation) is a lightweight data-interchange format. It is easy to read and write for humans and machines, making it widely used in web applications, APIs, and data exchange between systems.

 # Loading the JSON into a dataframe:

```python
import pandas as pd
arun = pd.read_json('data.js')
print(arun.to_string())
```

 # Dictionary as a JSON: if your JSON code is not in a file, but in a python dictionary, then you can do all below:

```
data = {
    "Duration":{
        "0":60,
        "1":60,
        "2":60,
        "3":45,
        "4":45,
        "5":60
        },
    "Pulse":{
        "0":110,
        "1":117,
        "2":103,
        "3":109,
        "4":117,
        "5":102
        },
    "Maxpulse":{
        "0":130,
        "1":145,
        "2":135,
        "3":175,
        "4":148,
        "5":127
        },
    "Calories":{
        "0":409.1,
        "1":479.0,
        "2":340.0,
        "3":282.4,
        "4":406.0,
        "5":300.5
        }
        }
arunnew = pd.DataFrame(data)
print(arunnew)
```

Viewing & Analyzing DataFrames

 # Viewing the data : one of the most used method for a quick overview of the dataframe is the head() method. this method returns the headers and a specified number of rows.

```
# here we will print the 1st 10 rows in the dataframe.
import pandas as pd
arun = pd.read_csv('data.csv')
print(arun.head(10))

                    # arun.head() -> this will return top 5 rows by default
# here we will print the last 10 rows in the dataframe.
import pandas as pd
arun = pd.read_csv('data.csv')
print(arun.tail(10))


# what if you want the information about the data in the dataframe: via info()
import pandas as pd
df = pd.read_csv('data.csv')
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 5.4 KB
None
```

Cleanina Data

cleaning data : it means fixing the bad data in your data set.

bad data could be: empty cell, data in a wrong format, duplicate data and wrong data.

emplty cell: it will give you wrong result always, we will have to remove the rows always that contain the bad data.

-> never make any changes in the original data, create a new file and to changes in that new file

```
# loading and reading the original dataframe
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
print(arun.to_string())

print("====================")

# here we will return a new data frame with no empty cell.

import pandas as pd
arun = pd.read_csv('dirtydata.csv')
arunnew = arun.dropna()
print(arunnew.to_string())
```

```python
# if at any case you want to change the original dataframe(but not recommended),
# than use the inplace=True argument. it will remove the rows containing the NULL(NaN(Not a Number)) values.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
arun.dropna(inplace=True)
print(arun.to_string())

print("=====================")

# replacing the empty value: we will use the fillna() method which will allow us
# to replace the empty cell with a value.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
arun.fillna( value: 130, inplace=True)
print(arun.to_string())

print("=====================")

# to replace only the empty value for one column , you need to specify the column name.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
arun["Calories"].fillna(130, inplace=True)  # replace NaN with 130 for Calories only
print(arun.to_string())
```

```python
# here we can also replace the empty cell using mean(), median() or mode().
#calculate the MEAN and replace the empty values with it.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
x = arun["Calories"].mean()
arun["Calories"].fillna(x, inplace=True)
print(arun.to_string())

print("=====================")

# calculate the MEDIAN and replace any empty values in it.:
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
x = arun["Calories"].median()             #median is better used when we are analysing trend
arun["Calories"].fillna(x, inplace=True)
print(arun.to_string())

print("==========10===========")

# calculate the MODE and replace the empty cell with it.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
x = arun["Calories"].mode()[0]  #[0]is the index,it will run a loop from 0 and find for the max repeating no.
arun["Calories"].fillna(x, inplace=True)
print(arun.to_string())
```

Cleaning Data of Wrong Format

Data in a wrong format:

to fix this problem, there are 2 ways: removing the rows or convert all the cells in the same format.

```
# Step 1: Load and display the original data
arun = pd.read_csv('dirtydata.csv')
print(arun.to_string())
print("======================")
```

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	NaN
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	20201226	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	NaN
29	60	'2020/12/29'	100	132	280.0

```python
# Step 2: Convert 'Date' column to datetime format
arun['Date'] = pd.to_datetime(arun['Date'], errors='coerce')
print(arun.to_string())
print("====================")
```

`errors='coerce'` : This will convert valid date strings to datetime objects and replace invalid formats with `NaT`.

```
21         60 2020-12-21    108         131      364.2
22         45        NaT    100         119      282.0
23         60 2020-12-23    130         101      300.0
24         45 2020-12-24    105         132      246.0
25         60 2020-12-25    102         126      334.5
26         60        NaT    100         120      250.0
27         60 2020-12-27     92         118      241.0
28         60 2020-12-28    103         132        NaN
```

```
# Convert Age to numeric, set errors='coerce' to handle invalid data
data['Age'] = pd.to_numeric(data['Age'], errors='coerce')
```

```python
# Step 3: Remove rows with null values in the 'Date' column
arun = pd.read_csv('dirtydata.csv')  # Reload original data
arun['Date'] = pd.to_datetime(arun['Date'], errors='coerce')
arun.dropna(subset=['Date'], inplace=True)
print(arun.to_string())
```

```
19         60 2020-12-19    103         123      323.0
20         45 2020-12-20     97         125      243.0
21         60 2020-12-21    108         131      364.2
23         60 2020-12-23    130         101      300.0
24         45 2020-12-24    105         132      246.0
25         60 2020-12-25    102         126      334.5
27         60 2020-12-27     92         118      241.0
28         60 2020-12-28    103         132        NaN
29         60 2020-12-29    100         132      280.0
```

`errors='coerce'` : Converts unparseable values to `NaT`.

wrong Data: its only a wrong data(age= 30.6)

```
# loading and reading the original dataframe
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
print(arun.to_string())
```

```
# Manually correcting the 'Duration' value for the 7th row (index 7)
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
arun.loc[7, 'Duration'] = 145  # Sets the value in the 'Duration' column at index 7 to 45
print(arun.to_string())  # Prints the entire DataFrame to see the updated data

# ==============================

# Loop through all rows in the 'Duration' column to correct invalid values
# If the 'Duration' value exceeds 120, it will be capped at 120
for i in arun.index:  # Iterates over all row indices in the DataFrame
    if arun.loc[i, "Duration"] > 120:  # Checks if the value in 'Duration' column is greater than 120
        arun.loc[i, "Duration"] = 120  # Updates the value to 120 if the condition is met

# Prints the DataFrame to verify the changes
print(arun.to_string())

# ==============================
```

```python
# Alternative approach: Removing rows with invalid data
# If 'Duration' exceeds 120, that row will be dropped from the dataset

import pandas as pd
arun = pd.read_csv('dirtydata.csv')
for i in arun.index:  # Iterates over all row indices
    if arun.loc[i, "Duration"] > 120:  # Checks for invalid data in the 'Duration' column
        arun.drop(i, inplace=True)  # Removes the entire row from the DataFrame

# Prints the modified DataFrame to confirm the rows have been removed
print(arun.to_string())
```

removing the duplicate values: 1st you need to discover the duplicate values via duplicated() method.

```python
# Step 1: Loading and reading the original data from a CSV file.
# The `read_csv()` method reads the dataset from 'dirtydata.csv' and creates a DataFrame named 'arun'.
# Using `to_string()` ensures the entire DataFrame is printed without truncation.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')
print(arun.to_string())  # Display the original dataset

# Step 2: Discovering duplicate rows in the dataset.
# The `duplicated()` method checks each row and returns True if the row is a duplicate of a previous row.
# It helps identify duplicate entries in the data without altering the original dataset.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')  # Reload the dataset
print(arun.duplicated())  # Output a boolean series indicating duplicate rows (True for duplicates)

# Step 3: Removing duplicate rows from the dataset.
# The `drop_duplicates()` method removes duplicate rows based on all columns by default.
# Setting `inplace=True` modifies the original DataFrame directly instead of creating a new dataframe.
import pandas as pd
arun = pd.read_csv('dirtydata.csv')  # Reload the dataset again
arun.drop_duplicates(inplace=True)  # Remove duplicates in place
print(arun.to_string())  # Display the cleaned dataset
```