Pandas

What is Pandas?

- **Pandas** is an open-source data analysis library written in Python.
- It leverages the power and speed of **NumPy** to make data analysis and preprocessing easy for data scientists.
- Provides **rich and highly robust data operations**.

Pandas Data Structures

Pandas has two main data structures:

- **Series** → A one-dimensional array with indexes. It stores a single column or row of data in a DataFrame.
- **DataFrame** → A tabular, spreadsheet-like structure where each row contains one or multiple columns.

Key Differences:

- **Series** → A one-dimensional labeled array capable of holding any type of data.
- **DataFrame** → A two-dimensional labeled data structure with columns of potentially different types of data.

In [1]:

```
# Import necessary Libraries
import numpy as np
import pandas as pd
```

In [2]:

```
# Create a dictionary
dict1 = {
    "Name" : ["Kushal", "Mukesh", "Ashok", "Shailendra"],
    "Marks" : [92, 34, 24, 17],
    "City" : ["Kathmandu", "Dhangadhi", "Dharan", "Chitwan"]
}
```

In [3]:

```
df = pd.DataFrame(dict1) # Converts our dictionary to data frame
df
```

Out[3]:

	Name	Marks	City
0	Kushal	92	Kathmandu
1	Mukesh	34	Dhangadhi
2	Ashok	24	Dharan
3	Shailendra	17	Chitwan

In [4]:
```
df.to_csv("friends.csv") # Create a csv file with the data
```

In [5]:
```
df.to_csv("no-index-friends.csv", index = False) # Create a csv file neglecting ind
```

In [6]:
```
df.head(2) # Displlays given number rows from the head side
```

Out[6]:

	Name	Marks	City
0	Kushal	92	Kathmandu
1	Mukesh	34	Dhangadhi

In [7]:
```
df.tail(2) # Displays given number of rows from tail side
```

Out[7]:

	Name	Marks	City
2	Ashok	24	Dharan
3	Shailendra	17	Chitwan

In [8]:
```
df.describe() # Statistical analysis for all the numerical columns
```

Out[8]:

	Marks
count	4.00000
mean	41.75000
std	34.21866
min	17.00000
25%	22.25000
50%	29.00000
75%	48.50000
max	92.00000

In [9]:
```
friends = pd.read_csv("friends.csv") # This will open our csv file by adding a inde
friends
```

Out[9]:

	Unnamed: 0	Name	Marks	City
0	0	Kushal	92	Kathmandu
1	1	Mukesh	34	Dhangadhi
2	2	Ashok	24	Dharan
3	3	Shailendra	17	Chitwan

In [10]:

```
friends = pd.read_csv("no-index-friends.csv") # Since we don't have index in csv it
friends
```

Out[10]:

	Name	Marks	City
0	Kushal	92	Kathmandu
1	Mukesh	34	Dhangadhi
2	Ashok	24	Dharan
3	Shailendra	17	Chitwan

In [11]:

```
type(friends) # It displays the type of the objects
```

Out[11]:

```
pandas.core.frame.DataFrame
```

In [12]:

```
friends.dtypes # Shows the datatypes of all the columns in data frame
```

Out[12]:

```
Name      object
Marks      int64
City      object
dtype: object
```

In [13]:

```
# Changing values in the data frame
friends["Marks"] # Provides only the selected column data
```

Out[13]:

```
0    92
1    34
2    24
3    17
Name: Marks, dtype: int64
```

In [14]:

```
friends["Marks"][0] # Provides only the data in selected index of selected column
```

Out[14]:

```
np.int64(92)
```

In [15]:

```
# So you can change the data at that index
friends["Marks"][0] = 98 # This will be confusing for python interpreter whether to
```

```
C:\Users\kusha\AppData\Local\Temp\ipykernel_24672\82900966.py:2: SettingWithCopyWarn
ing:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  friends["Marks"][0] = 98 # This will be confusing for python interpreter whether t
o return copy or view so we use loc() function
```

In [16]:

```
# Now can see that the value has been changed
friends
```

Out[16]:

	Name	Marks	City
0	Kushal	98	Kathmandu
1	Mukesh	34	Dhangadhi
2	Ashok	24	Dharan
3	Shailendra	17	Chitwan

In [17]:

```
# You can update it to the csv file or create a new csv file
# Note that use same name if you want to update and different name if you want to c
friends.to_csv("no-index-friends.csv", index = False)
# index = False is necessary because it avoids addition of an extra index column in
```

In [18]:

```
# Changing the index of the data frame
friends.index = ["first", "second", "third", "fourth"]
friends
```

Out[18]:

	Name	Marks	City
first	Kushal	98	Kathmandu
second	Mukesh	34	Dhangadhi
third	Ashok	24	Dharan
fourth	Shailendra	17	Chitwan

In [19]:

```
ser = pd.Series(np.random.rand(34)) # Create a series with random numbers
ser
```

Out[19]:

```
0     0.360879
1     0.762553
2     0.904318
3     0.281279
4     0.159259
5     0.691495
6     0.883009
7     0.472963
8     0.313954
9     0.233485
10    0.170632
11    0.501466
12    0.576821
13    0.400094
14    0.976389
15    0.619292
16    0.987180
17    0.515985
18    0.322008
19    0.294672
20    0.320987
21    0.421319
22    0.217760
23    0.519052
24    0.367007
25    0.971693
26    0.554057
27    0.939355
28    0.352071
29    0.685393
30    0.853083
31    0.197149
32    0.611799
33    0.107569
dtype: float64
```

In [20]:

```
# Create a Data Frame with random numbers
new_df = pd.DataFrame(np.random.rand(334, 5), index = np.arange(334))
```

In [21]:

```
new_df.index # Shows all the index in the data frame
```

Out[21]:

```
Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
       ...
       324, 325, 326, 327, 328, 329, 330, 331, 332, 333],
      dtype='int64', length=334)
```

In [22]:

```
new_df.columns # Show all the columns in the data frame
```

Out[22]:

```
RangeIndex(start=0, stop=5, step=1)
```

In [23]:

```
new_df.to_numpy() # Converts data frame to numpy object
```

Out[23]:
```
array([[0.42550724, 0.7916049 , 0.20794834, 0.99392538, 0.2920232 ],
       [0.96222715, 0.81145081, 0.43916541, 0.85014979, 0.93406513],
       [0.28807486, 0.64877028, 0.37987549, 0.77244959, 0.14713672],
       ...,
       [0.11327033, 0.88943813, 0.84111197, 0.51922324, 0.51869155],
       [0.2435048 , 0.15433019, 0.18761044, 0.08846024, 0.47176376],
       [0.5880696 , 0.07941395, 0.81143397, 0.93159245, 0.0161998 ]],
      shape=(334, 5))
```

In [24]:
```python
new_df.T # It transpose the data frame like matrices
```

Out[24]:

	0	1	2	3	4	5	6	7	8
0	0.425507	0.962227	0.288075	0.251557	0.507984	0.042327	0.953632	0.369517	0.695066
1	0.791605	0.811451	0.648770	0.046233	0.588743	0.115894	0.143614	0.364205	0.334907
2	0.207948	0.439165	0.379875	0.403854	0.817332	0.375170	0.509991	0.324721	0.388108
3	0.993925	0.850150	0.772450	0.154160	0.218569	0.280429	0.778723	0.274897	0.827896
4	0.292023	0.934065	0.147137	0.298002	0.822953	0.909996	0.908635	0.704739	0.219064

5 rows × 334 columns

In [25]:
```python
new_df.sort_index(axis = 0, ascending=False) # Sort the index of data frame: axis =
```

Out[25]:

	0	1	2	3	4
333	0.588070	0.079414	0.811434	0.931592	0.016200
332	0.243505	0.154330	0.187610	0.088460	0.471764
331	0.113270	0.889438	0.841112	0.519223	0.518692
330	0.344206	0.034836	0.027302	0.710629	0.676302
329	0.927718	0.160133	0.311260	0.212767	0.373772
...	...	...	...	...	...
4	0.507984	0.588743	0.817332	0.218569	0.822953
3	0.251557	0.046233	0.403854	0.154160	0.298002
2	0.288075	0.648770	0.379875	0.772450	0.147137
1	0.962227	0.811451	0.439165	0.850150	0.934065
0	0.425507	0.791605	0.207948	0.993925	0.292023

334 rows × 5 columns

In [26]:
```python
type(new_df[0]) # The combination of Series is DataFrame
```

Out[26]:

```
pandas.core.series.Series
```

In [27]:

```
new_df2 = new_df # Here new_df2 is just a view of new_df,
# If you change new_df2 then new_df will also change because both are pointing same
new_df2[0][0] = 0
new_df # The element in new_df will also be changed
```

```
C:\Users\kusha\AppData\Local\Temp\ipykernel_24672\74676621.py:3: FutureWarning: Chai
nedAssignmentError: behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in certain c
ases, but when using Copy-on-Write (which will become the default behaviour in panda
s 3.0) this will never work to update the original DataFrame or Series, because the
intermediate object on which we are setting values will behave as a copy.
A typical example is when you are setting values in a column of a DataFrame, like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment in a si
ngle step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy

  new_df2[0][0] = 0
```

Out[27]:

	0	1	2	3	4
0	0.000000	0.791605	0.207948	0.993925	0.292023
1	0.962227	0.811451	0.439165	0.850150	0.934065
2	0.288075	0.648770	0.379875	0.772450	0.147137
3	0.251557	0.046233	0.403854	0.154160	0.298002
4	0.507984	0.588743	0.817332	0.218569	0.822953
...	...	...	...	...	...
329	0.927718	0.160133	0.311260	0.212767	0.373772
330	0.344206	0.034836	0.027302	0.710629	0.676302
331	0.113270	0.889438	0.841112	0.519223	0.518692
332	0.243505	0.154330	0.187610	0.088460	0.471764
333	0.588070	0.079414	0.811434	0.931592	0.016200

334 rows × 5 columns

In [28]:

```
new_df3 = new_df.copy() # Creates a copy.
# Now changes will not affect new_df
new_df3[0][0] = 0.5
new_df # No change in original while changing copy
```

```
C:\Users\kusha\AppData\Local\Temp\ipykernel_24672\924715784.py:3: FutureWarning: Cha
inedAssignmentError: behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in certain c
ases, but when using Copy-on-Write (which will become the default behaviour in panda
s 3.0) this will never work to update the original DataFrame or Series, because the
intermediate object on which we are setting values will behave as a copy.
A typical example is when you are setting values in a column of a DataFrame, like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment in a si
ngle step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy

  new_df3[0][0] = 0.5
```

Out[28]:

	0	1	2	3	4
0	0.000000	0.791605	0.207948	0.993925	0.292023
1	0.962227	0.811451	0.439165	0.850150	0.934065
2	0.288075	0.648770	0.379875	0.772450	0.147137
3	0.251557	0.046233	0.403854	0.154160	0.298002
4	0.507984	0.588743	0.817332	0.218569	0.822953
...	...	...	...	...	...
329	0.927718	0.160133	0.311260	0.212767	0.373772
330	0.344206	0.034836	0.027302	0.710629	0.676302
331	0.113270	0.889438	0.841112	0.519223	0.518692
332	0.243505	0.154330	0.187610	0.088460	0.471764
333	0.588070	0.079414	0.811434	0.931592	0.016200

334 rows × 5 columns

In [29]:

```
new_df.loc[0,0] = 654 # Using loc() function is a proper way to change objects in D
new_df.head()
```

Out[29]:

	0	1	2	3	4
0	654.000000	0.791605	0.207948	0.993925	0.292023
1	0.962227	0.811451	0.439165	0.850150	0.934065
2	0.288075	0.648770	0.379875	0.772450	0.147137
3	0.251557	0.046233	0.403854	0.154160	0.298002
4	0.507984	0.588743	0.817332	0.218569	0.822953

In [30]:

```
new_df.columns = list("ABCDE") # Change columns name to A B C D E
new_df
```

Out[30]:

	A	B	C	D	E
0	654.000000	0.791605	0.207948	0.993925	0.292023
1	0.962227	0.811451	0.439165	0.850150	0.934065
2	0.288075	0.648770	0.379875	0.772450	0.147137
3	0.251557	0.046233	0.403854	0.154160	0.298002
4	0.507984	0.588743	0.817332	0.218569	0.822953
...	...	...	...	...	...
329	0.927718	0.160133	0.311260	0.212767	0.373772
330	0.344206	0.034836	0.027302	0.710629	0.676302
331	0.113270	0.889438	0.841112	0.519223	0.518692
332	0.243505	0.154330	0.187610	0.088460	0.471764
333	0.588070	0.079414	0.811434	0.931592	0.016200

334 rows × 5 columns

In [31]:

```
#new_df.loc[0, 0] = 100
# Will create a new column 0 with all other elements NaN as we have changed columns

# So we can use loc() function to change the value of the column A
# Here we are changing the value of first row in column A to 90
new_df.loc[0, "A"] = 90
new_df
```

Out[31]:

	A	B	C	D	E
0	90.000000	0.791605	0.207948	0.993925	0.292023
1	0.962227	0.811451	0.439165	0.850150	0.934065
2	0.288075	0.648770	0.379875	0.772450	0.147137
3	0.251557	0.046233	0.403854	0.154160	0.298002
4	0.507984	0.588743	0.817332	0.218569	0.822953
...	...	...	...	...	...
329	0.927718	0.160133	0.311260	0.212767	0.373772
330	0.344206	0.034836	0.027302	0.710629	0.676302
331	0.113270	0.889438	0.841112	0.519223	0.518692
332	0.243505	0.154330	0.187610	0.088460	0.471764
333	0.588070	0.079414	0.811434	0.931592	0.016200

334 rows × 5 columns

In [32]:

```
new_df = new_df.drop("E", axis = 1) # Remove E column
# Always remember axis = 0 for row and axis = 1 for column
new_df
```

Out[32]:

	A	B	C	D
0	90.000000	0.791605	0.207948	0.993925
1	0.962227	0.811451	0.439165	0.850150
2	0.288075	0.648770	0.379875	0.772450
3	0.251557	0.046233	0.403854	0.154160
4	0.507984	0.588743	0.817332	0.218569
...	...	...	...	...
329	0.927718	0.160133	0.311260	0.212767
330	0.344206	0.034836	0.027302	0.710629
331	0.113270	0.889438	0.841112	0.519223
332	0.243505	0.154330	0.187610	0.088460
333	0.588070	0.079414	0.811434	0.931592

334 rows × 4 columns

In [33]:

```
new_df.loc[[1, 2], ["C", "D"]] # Locate the selected rows and columns
```

Out[33]:

	C	D
1	0.439165	0.85015
2	0.379875	0.77245

In [34]:

```
#If we need all row and columns we use : on the requird place
new_df.loc[[1, 2], :]
```

Out[34]:

	A	B	C	D
1	0.962227	0.811451	0.439165	0.85015
2	0.288075	0.648770	0.379875	0.77245

In [35]:

```
new_df.loc[(new_df["A"] > 0.3)] # Locate the rows in which column A has vlaue > 0.3
```

Out[35]:

	A	B	C	D
0	90.000000	0.791605	0.207948	0.993925
1	0.962227	0.811451	0.439165	0.850150
4	0.507984	0.588743	0.817332	0.218569
6	0.953632	0.143614	0.509991	0.778723
7	0.369517	0.364205	0.324721	0.274897
...	...	...	...	...
327	0.419485	0.014310	0.817724	0.889772
328	0.796684	0.023844	0.061294	0.631596
329	0.927718	0.160133	0.311260	0.212767
330	0.344206	0.034836	0.027302	0.710629
333	0.588070	0.079414	0.811434	0.931592

225 rows × 4 columns

In [36]:

```
# Locate the rows in which column A has vlaue > 0.3 and column C has value > 0.1
new_df.loc[(new_df["A"] > 0.3) & (new_df["C"] > 0.1)]
```

Out[36]:

	A	B	C	D
0	90.000000	0.791605	0.207948	0.993925
1	0.962227	0.811451	0.439165	0.850150
4	0.507984	0.588743	0.817332	0.218569
6	0.953632	0.143614	0.509991	0.778723
7	0.369517	0.364205	0.324721	0.274897
...	...	...	...	...
322	0.972578	0.752600	0.841395	0.002945
323	0.437006	0.209989	0.980634	0.122960
327	0.419485	0.014310	0.817724	0.889772
329	0.927718	0.160133	0.311260	0.212767
333	0.588070	0.079414	0.811434	0.931592

204 rows × 4 columns

In [37]:

```
new_df.iloc[0, 3] # ilock() takes value of [i, j] just like in matrices
```

Out[37]:

```
np.float64(0.9939253763603781)
```

In [38]:

```
new_df.loc[0, "D"] # loc() takes values of [row, column] in Data Frame
```

Out[38]:

```
np.float64(0.9939253763603781)
```

In [39]:

```
# iloc() can be used instead of loc() every where just remember that it takes [i, j
new_df.iloc[[0, 5], [1, 2]]
```

Out[39]:

	B	C
0	0.791605	0.207948
5	0.115894	0.375170

In [40]:

```
new_df.drop(["A", "C"], axis = 1) # Delete the selected columns as axis = 1 (Return
# Remember that this will not change the original Data Frame until we same it like:
```

Out[40]:

	B	D
0	0.791605	0.993925
1	0.811451	0.850150
2	0.648770	0.772450
3	0.046233	0.154160
4	0.588743	0.218569
...	...	...
329	0.160133	0.212767
330	0.034836	0.710629
331	0.889438	0.519223
332	0.154330	0.088460
333	0.079414	0.931592

334 rows × 2 columns

In [41]:

```
# If we use inplace=True it will change the original Data Frame
new_df.drop(["C", "D"], axis = 1, inplace=True)
```

In [42]:

```
new_df
```

Out[42]:

	A	B
0	90.000000	0.791605
1	0.962227	0.811451
2	0.288075	0.648770
3	0.251557	0.046233
4	0.507984	0.588743
...	...	...
329	0.927718	0.160133
330	0.344206	0.034836
331	0.113270	0.889438
332	0.243505	0.154330
333	0.588070	0.079414

334 rows × 2 columns

In [43]:

```
# Lets remove some rows from middle
new_df.drop([i for i in range(6, 330)], axis=0, inplace=True)
```

In [44]:

```
new_df
```

Out[44]:

	A	B
0	90.000000	0.791605
1	0.962227	0.811451
2	0.288075	0.648770
3	0.251557	0.046233
4	0.507984	0.588743
5	0.042327	0.115894
330	0.344206	0.034836
331	0.113270	0.889438
332	0.243505	0.154330
333	0.588070	0.079414

In [45]:

```
# We need to reset the index now
# new_df.reset_index() # This will add a new column named index so drop=True is nec
# new_df.reset_index(drop=True) #This will not change the original Data Frame so in
new_df.reset_index(drop=True, inplace=True)
```

In [46]:

```
new_df
```

Out[46]:

	A	B
0	90.000000	0.791605
1	0.962227	0.811451
2	0.288075	0.648770
3	0.251557	0.046233
4	0.507984	0.588743
5	0.042327	0.115894
6	0.344206	0.034836
7	0.113270	0.889438
8	0.243505	0.154330
9	0.588070	0.079414

In [47]:

```
new_df["B"].isnull() # Returns True if the null and False otherwise
```

Out[47]:

```
0    False
1    False
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
Name: B, dtype: bool
```

In [48]:

```
# new_df["A"] = None # Make all the values in A column null
new_df.loc[:, "A"] = None # Good practice
```

```
C:\Users\kusha\AppData\Local\Temp\ipykernel_24672\2896297378.py:2: FutureWarning: Se
tting an item of incompatible dtype is deprecated and will raise in a future error o
f pandas. Value 'None' has dtype incompatible with float64, please explicitly cast t
o a compatible dtype first.
  new_df.loc[:, "A"] = None # Good practice
```

In [49]:

```
new_df
```

Out[49]:

	A	B
0	None	0.791605
1	None	0.811451
2	None	0.648770
3	None	0.046233
4	None	0.588743
5	None	0.115894
6	None	0.034836
7	None	0.889438
8	None	0.154330
9	None	0.079414

In [50]:

```
new_df["B"].isnull() # All the values in B columns are changed to null
```

Out[50]:

```
0    False
1    False
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
Name: B, dtype: bool
```

In [51]:

```
df_new = pd.DataFrame( {"name" : ['Alfred', 'Batman', 'Catwoman'],
                        "toy" : [np.nan, 'Batmobile', 'Bullwhip'],
                        "born" : [pd.NaT, pd.Timestamp("1940-04-25"), pd.NaT]})
df_new
```

Out[51]:

	name	toy	born
0	Alfred	NaN	NaT
1	Batman	Batmobile	1940-04-25
2	Catwoman	Bullwhip	NaT

In [52]:

```
df_new.dropna() # Removes all rows containing na because default axis=0
```

Out[52]:

	name	toy	born
1	Batman	Batmobile	1940-04-25

In [53]:

```
df_new.dropna(how="all") # Removes rows if all values in it are na because default
```

Out[53]:

	name	toy	born
0	Alfred	NaN	NaT
1	Batman	Batmobile	1940-04-25
2	Catwoman	Bullwhip	NaT

In [54]:

```
# You can also choose axis for deletion
df_new.dropna(axis=1)
```

Out[54]:

	name
0	Alfred
1	Batman
2	Catwoman

In [55]:

```
# Lets make some duplicates in Data Frame
df_new.loc[2, "name"] = "Alfred"
df_new
```

Out[55]:

	name	toy	born
0	Alfred	NaN	NaT
1	Batman	Batmobile	1940-04-25
2	Alfred	Bullwhip	NaT

In [56]:

```
# Lets remove the duplicates now
df_new.drop_duplicates(subset=['name'], keep= 'last')
# keep = 'first' is default
# keep = 'last' removes all other than last
# keep = False removes all
```

Out[56]:

	name	toy	born
1	Batman	Batmobile	1940-04-25
2	Alfred	Bullwhip	NaT

In [57]:

```
df_new.shape # Provides information about number of rows and columns
```

Out[57]:

```
(3, 3)
```

In [58]:

```
df_new.info() # Displays information about the Data Frame
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   name    3 non-null      object
 1   toy     2 non-null      object
 2   born    1 non-null      datetime64[ns]
dtypes: datetime64[ns](1), object(2)
memory usage: 204.0+ bytes
```

In [59]:

```
df_new['toy'].value_counts(dropna=False) # Displays counts of the elements in colum
# if dropna=False it displays without removing NaN values
# if dropna=True it displays after removing NaN values
```

Out[59]:

```
toy
NaN          1
Batmobile    1
Bullwhip     1
Name: count, dtype: int64
```

In [60]:

```
df_new.notnull() # Returns True if value is not null else return False
```

Out[60]:

	name	toy	born
0	True	False	False
1	True	True	True
2	True	True	False