Pandas is an open-source, BSD-licensed Python library. Python package providing fast, flexible, handy, and expressive data structures tool designed to make working with 'relationa' or 'labeled' data both easy and intuitive.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data with row and column labels
- Any other form of observational / statistical data sets.

```
# Dataframe from dict
import pandas as pd

# Python dict object
student_dict = {'Name': ['Joe', 'Nat'], 'Age': [20, 21], 'Marks': [85.10, 77.80]}
print(student_dict)

# Create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)


df = pd.DataFrame({'A': [1, 2, 3], 'B': [True, True, False],
                   'C': [0.496714, -0.138264, 0.647689]},
                  index=['a', 'b', 'c'])  # also this weird index thing
df


# Create DataFrame from list

# Create list
fruits_list = ['Apple', 'Banana', 'Orange','Mango']
print(fruits_list)

# Create DataFrame from list
fruits_df = pd.DataFrame(fruits_list, columns=['Fruits'])
print(fruits_df)


# Create DataFrame from hierarchical lists as rows

# Create list
fruits_list = [['Apple', 'Banana', 'Orange', 'Mango'],[120, 40, 80, 500]]
print(fruits_list)

# Create DataFrame from list
fruits_df = pd.DataFrame(fruits_list)
print(fruits_df)


# Create DataFrame from Hierarchical lists as columns

# Create list
fruits_list = [['Apple', 'Banana', 'Orange', 'Mango'],[120, 40, 80, 500]]
print(fruits_list)

# Create DataFrame from list
fruits_df = pd.DataFrame(fruits_list).transpose()
print(fruits_df)


# Create DataFrame from multiple lists

# Create multiple lists
fruits_list = ['Apple', 'Banana', 'Orange', 'Mango']
price_list = [120, 40, 80, 500]

# Create DataFrame
fruits_df = pd.DataFrame(list(zip(fruits_list, price_list )), columns = ['Name', 'Price'])
print(fruits_df)
```

Indexing: Our first improvement over numpy arrays is labeled indexing. We can select subsets by column, row, or both.

- Use [] for selecting columns

- Use .loc[row_lables, column_labels] for label-based indexing
- Use .iloc[row_positions, column_positions] for positional index

```
# Single column, reduces to a Series
df['A']
```

```
cols = ['A', 'C']
df[cols]
```

```
# For row-wise selection, use the special .loc accessor.
df.loc[['a', 'b']]
```

```
df.loc['a':'b']
```

```
# Sometimes, you'd rather slice by position instead of label. .iloc has you covered:

df.iloc[[0, 1]]
```

```
df.iloc[:2]
```

```
# Selecting a single row and single column

df.loc['a', 'B']
```

```
df.loc['a':'b', ['A', 'C']]
```

Dataframe from CSV: In the field of Data Science, CSV files are used to store large datasets. To efficiently analyze such datasets, we need to convert them into pandas DataFrame.

```
data = pd.read_csv("/content/sample_data/stockprice_data.csv")
data
```

```
# Convert Pandas Data Frame in to Pandas Series

data1 = data.iloc[:,2]
data1
```

```
type(data)
```

```
type(data1)
```

To customize the display of DataFrame while printing

When we display the DataFrame using print() function by default, it displays 10 rows (top 5 and bottom 5). Sometimes we may need to show more or lesser rows than the default view of the DataFrame.

We can change the setting by using pd.options or pd.set_option() functions. Both can be used interchangeably.

```
# just give the name of file only if the file is in the same folder.

cars = pd.read_csv("/content/sample_data/automobile_data.csv")
print(cars)
```

```
# Setting maximum rows to be shown
pd.options.display.max_rows = 20

# Setting minimum rows to be shown
pd.set_option("display.min_rows", 10)

# Print DataFrame
print(cars)
```

DataFrame metadata

Sometimes we need to get metadata of the DataFrame and not the content inside it. Such metadata information is useful to understand the DataFrame as it gives more details about the DataFrame that we need to process.

```
# get dataframe info
cars.info()
```

```
# get dataframe description
cars.describe()
```

DataFrame Attributes

DataFrame has provided many built-in attributes. Attributes do not modify the underlying data, unlike functions, but it is used to get more details about the DataFrame.

Following are majorly used attributes of the DataFrame:

Attribute	Description
`DataFrame.index`	It gives the Range of the row index
`DataFrame.columns`	It gives a list of column labels
`DataFrame.dtypes`	It gives column names and their data type
`DataFrame.values`	It gives all the rows in DataFrame
`DataFrame.empty`	It is used to check if the DataFrame is empty
`DataFrame.size`	It gives a total number of values in DataFrame
`DataFrame.shape`	It a number of rows and columns in DataFrame

```
# Create DataFrame from dict
student_dict = {'Name': ['Joe', 'Nat', 'Harry'], 'Age': [20, 21, 19], 'Marks': [85.10, 77.80, 91.54]}

student_df = pd.DataFrame(student_dict)

print("DataFrame : ", student_df)

print("DataFrame Index : ", student_df.index)

print("DataFrame Columns : ", student_df.columns)

print("DataFrame Column types : ", student_df.dtypes)

print("DataFrame is empty? : ", student_df.empty)

print("DataFrame Shape : ", student_df.shape)

print("DataFrame Size : ", student_df.size)

print("DataFrame Values : ", student_df.values)
```

DataFrame selection

While dealing with the vast data in DataFrame, a data analyst always needs to select a particular row or column for the analysis. In such cases, functions that can choose a set of rows or columns like top rows, bottom rows, or data within an index range play a significant role.

Following are the functions that help in selecting the subset of the DataFrame:

Attribute	Description
`DataFrame.head(n)`	It is used to select top 'n' rows in DataFrame.
`DataFrame.tail(n)`	It is used to select bottom 'n' rows in DataFrame.
`DataFrame.at`	It is used to get and set the particular value of DataFrame using row and column labels.
`DataFrame.iat`	It is used to get and set the particular value of DataFrame using row and column index positions.
`DataFrame.get(key)`	It is used to get the value of a key in DataFrame where Key is the column name.
`DataFrame.loc()`	It is used to select a group of data based on the row and column labels. It is used for slicing and filtering of the DataFrame.
`DataFrame.iloc()`	It is used to select a group of data based on the row and column index position. Use it for slicing and filtering the DataFrame.

```
# display dataframe
print("DataFrame : ", student_df)

# select top 2 rows
print(student_df.head(2))

# select bottom 2 rows
print(student_df.tail(2))

# select value at row index 0 and column 'Name'
print(student_df.at[0, 'Name'])

# select value at first row and first column
print(student_df.iat[0, 0])

# select values of 'Name' column
print(student_df.get('Name'))

# select values from row index 0 to 2 and 'Name' column
print(student_df.loc[0:2, ['Name']])

# select values from row index 0 to 2(exclusive) and column position 0 to 2(exclusive)
print(student_df.iloc[0:2, 0:2])


# insert columns

# Create DataFrame from dict
student_dict = {'Name': ['Joe', 'Nat', 'Harry'], 'Age': [20, 21, 19], 'Marks': [85.10, 77.80, 91.54]}
student_df = pd.DataFrame(student_dict)
print(student_df)

# insert new column in dataframe and display
student_df.insert(loc=2, column="Class", value='A')
print(student_df)


# delete column from dataframe
student_df = student_df.drop(columns='Age')
print(student_df)
```

Apply condition

We may need to update the value in the DataFrame based on some condition. DataFrame.where()function is used to replace the value of DataFrame, where the condition is False.

Syntax:

where(filter, other=new_value)

- It applies the filter condition on all the rows in the DataFrame, as follows:
- If the filter condition returns False, then it updates the row with the value specified in other parameter.
- If the filter condition returns True, then it does not update the row.

```
# Create DataFrame from dict
student_dict = {'Name': ['Joe', 'Nat', 'Harry'], 'Age': [20, 21, 19], 'Marks': [85.10, 77.80, 91.54]}

student_df = pd.DataFrame(student_dict)
print(student_df)
```

```
# Define filter condition
filter = student_df['Marks'] > 80

student_df['Marks'].where(filter, other=0, inplace=True)
print(student_df)
```

DataFrame Join

In most of the use cases of Data Analytics, data gathered from multiple sources, and we need to combine that data for further analysis. In such instances, join and merge operations are required.

DataFrame.join() function is used to join one DataFrame with another DataFrame as df1.join(df2)

```
# create dataframe from dict
student_dict = {'Name': ['Joe', 'Nat'], 'Age': [20, 21]}
student_df = pd.DataFrame(student_dict)
print(student_df)

# create dataframe from dict
marks_dict = {'Marks': [85.10, 77.80]}
marks_df = pd.DataFrame(marks_dict)
print(marks_df)

# join dfs
joined_df = student_df.join(marks_df)
print(joined_df)
```

DataFrame GroupBy

GroupBy operation means splitting the data and then combining them based on some condition. Large data can be divided into logical groups to analyze it.

DataFrame.groupby() function groups the DataFrame row-wise or column-wise based on the condition.

```
# Create DataFrame from dict
student_dict = {'Name': ['Joe', 'Nat', 'Harry'], 'Class': ['A', 'B', 'A'], 'Marks': [85.10, 77.80, 91.54]}
student_df = pd.DataFrame(student_dict)
print(student_df)

# apply group by
student_df = student_df.groupby('Class').mean()
print(student_df)
```

DataFrame Iteration

DataFrame iteration means visiting each element in the DataFrame one by one. While analyzing a DataFrame, we may need to iterate over each row of the DataFrame.

There are multiple ways to iterate a DataFrame. We will see the function DataFrame.iterrows(), which can loop a DataFrame row-wise. It returns the index and row of the DataFrame in each iteration of the for a loop.

```
# Create DataFrame from dict
student_dict = {'Name': ['Joe', 'Nat'], 'Age': [20, 21], 'Marks': [85, 77]}
student_df = pd.DataFrame(student_dict)

# Iterate all the rows of DataFrame
for index, row in student_df.iterrows():
    print(index, row)
```

Importing Data

Operator	Description
`pd.read_csv(filename)`	From a CSV file
`pd.read_table(filename)`	From a delimited text file (like TSV)
`pd.read_excel(filename)`	From an Excel file
`pd.read_sql(query, connection_object)`	Read from a SQL table/database
`pd.read_json(json_string)`	Read from a JSON formatted string, URL or file.
`pd.read_html(url)`	Parses an html URL, string or file and extracts tables to a list of dataframes
`pd.read_clipboard()`	Takes the contents of your clipboard and passes it to read_table()
`pd.DataFrame(dict)`	From a dict, keys for columns names, values for data as lists

Exporting Data

Operator	Description
`df.to_csv(filename)`	Write to a CSV file
`df.to_excel(filename)`	Write to an Excel file
`df.to_sql(table_name, connection_object)`	Write to a SQL table
`df.to_json(filename)`	Write to a file in JSON format

```
cars
```

	index	company	body-style	wheel-base	length	engine-type	num-of-cylinders	horsepower	average-mileage	price
0	0	alfa-romero	convertible	88.6	168.8	dohc	four	111	21	13495.0
1	1	alfa-romero	convertible	88.6	168.8	dohc	four	111	21	16500.0
2	2	alfa-romero	hatchback	94.5	171.2	ohcv	six	154	19	16500.0
3	3	audi	sedan	99.8	176.6	ohc	four	102	24	13950.0
4	4	audi	sedan	99.4	176.6	ohc	five	115	18	17450.0
...	...	...	...	...	...	...	...	...	...	...
56	81	volkswagen	sedan	97.3	171.7	ohc	four	85	27	7975.0
57	82	volkswagen	sedan	97.3	171.7	ohc	four	52	37	7995.0
58	86	volkswagen	sedan	97.3	171.7	ohc	four	100	26	9995.0
59	87	volvo	sedan	104.3	188.8	ohc	four	114	23	12940.0
60	88	volvo	wagon	104.3	188.8	ohc	four	114	23	13415.0

61 rows × 10 columns

Exercises:

1. From the given automobile dataset print the first and last five rows
2. Clean the dataset and update the CSV file
3. Find the most and least expensive car company name
4. Print All BMW Cars details
5. Count total cars per body-style
6. Find each company's Higesht price and Lowest price cars
7. Find the average mileage of each car making company
8. Sort all cars by Price horsepower

9. Select the data in rows [13, 24, 58] and in columns ['animal', 'body-style','price']
10. Select only the rows where the wheel-base is between 85 and 90, both inclusive.
11. Change the price in row 59 to 13240.
12. Calculate the sum of all prices.
13. Append a new row 30 to the data frame with your choice of values for each column. Then delete that row to return the original DataFrame.