Pandas Python Library

Notes

Harsh Choudhary

Pandas Python Library

Pandas Introduction

Pandas is open-source Python library which is used for data manipulation and analysis. It consist of data structures and functions to perform efficient operations on data. It is well-suited for working with **tabular data** such as **spreadsheets** or **SQL tables**. It is used in data science because it works well with other important libraries. **It is built on top of the NumPy library** as it makes easier to manipulate and analyze. Pandas is used in other libraries such as:

- **Matplotlib** for plotting graphs
- **SciPy** for statistical analysis
- **Scikit-learn** for machine learning algorithms.
- It uses many functionalities provided by **NumPy library.**

Here is a various tasks that we can do using Pandas:

- **Data Cleaning, Merging and Joining**: Clean and combine data from multiple sources, handling inconsistencies and duplicates.
- **Handling Missing Data**: Manage missing values (NaN) in both floating and non-floating point data.
- **Column Insertion and Deletion**: Easily add, remove or modify columns in a DataFrame.
- **Group By Operations**: Use "split-apply-combine" to group and analyze data.
- **Data Visualization**: Create visualizations with Matplotlib and Seaborn, integrated with Pandas.

Getting Started with Pandas
Let's see how to start working with the Python Pandas library:

Installing Pandas
First step in working with Pandas is to ensure whether it is installed in the system or not. If not then we need to install it on our system using the **pip command**.
pip install pandas
For more reference, take a look at this article on installing pandas.

Importing Pandas
After the Pandas have been installed in the system we need to import the library. This module is imported using:
import pandas as pd
Note: **pd** is just an alias for Pandas. It's not required but using it makes the code shorter when calling methods or properties.

Data Structures in Pandas Library
Pandas provide two data structures for manipulating data which are as follows:
1. Pandas Series
A Pandas Series is one-dimensional labeled array capable of holding data of any type (integer, string, float, Python objects etc.). The axis labels are collectively called **indexes**.
Pandas Series is created by loading the datasets from existing storage which can be a SQL database, a CSV file or an Excel file. It can be created from lists, dictionaries, scalar values, etc.
Example: Creating a series using the Pandas Library.

```
import pandas as pd
import numpy as np

ser = pd.Series()
print("Pandas Series: ", ser)

data = np.array(['g', 'e', 'e', 'k', 's'])

ser = pd.Series(data)
print("Pandas Series:\n", ser)
```

2. Pandas DataFrame

Pandas DataFrame is a two-dimensional data structure with labeled axes (rows and columns). It is created by loading the datasets from existing storage which can be a SQL database, a CSV file or an Excel file. It can be created from lists, dictionaries, a list of dictionaries etc.

Example: Creating a DataFrame Using the Pandas Library

```
import pandas as pd

df = pd.DataFrame()
print(df)

lst = ['Tech', 'For', 'Tech', 'is', 'portal', 'for', 'Tech']

df = pd.DataFrame(lst)
print(df)
```

How to Install Pandas in Python?

Pandas in Python is a package that is written for data analysis and manipulation. Pandas offer various operations and data structures to perform numerical data manipulations and time series. Pandas is an open-source library that is built over Numpy libraries. Pandas library is known for its high productivity and high performance. Pandas are popular because they make importing and analyzing data much easier. Pandas programs can be written on any plain text editor like **Notepad, notepad++,** or anything of that sort and saved with a **.py** extension.

To begin with Install Pandas in Python, write Pandas Codes, and perform various intriguing and useful operations, one must have Python installed on their System.

Check if Python is Already Present

To check if your device is pre-installed with Python or not, just go to the **Command line**(search for **cmd** in the Run dialog(**+ R**). Now run the following command:

```
python --version
```

If Python is already installed, it will generate a message with the Python version available else install Python, for installing please visit:

How to Install Python on Windows or Linux and PIP.

Python version

Pandas can be installed in multiple ways on Windows, Linux, and MacOS. Various ways are listed below:

Import Pandas in Python

Now, that we have installed pandas on the system. Let's see how we can import it to make use of it. For this, go to a Jupyter Notebook or open a Python file, and write the following code:

```
import pandas as pd
```

Here, pd is referred to as an alias to the Pandas, which will help us in optimizing the code.

How to Install or Download Python Pandas

Pandas can be installed in multiple ways on Windows, Linux and MacOS. Various different ways are listed below:

Install Pandas on Windows

Python Pandas can be installed on Windows in two ways:

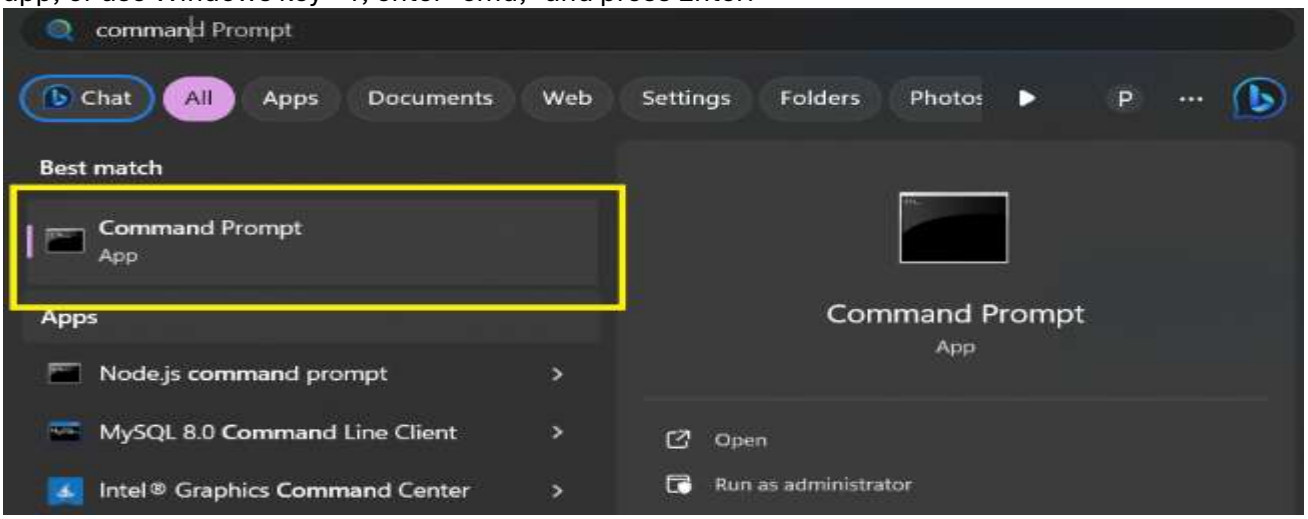
- Using pip
- Using Anaconda

Install Pandas using pip

PIP is a package management system used to install and manage software packages/libraries written in Python. These files are stored in a large "online repository" termed as Python Package Index (PyPI).

Step 1 : Launch Command Prompt

To open the Start menu, press the Windows key or click the Start button. To access the Command Prompt, type "cmd" in the search bar, click the displayed
app, or use Windows key + r, enter "cmd," and press Enter.



Command Prompt

Step 2 : Run the Command

Pandas can be installed using PIP by use of the following command in Command Prompt.

```
pip install pandas
```

Install Pandas using Anaconda

Anaconda is open-source software that contains Jupyter, spyder, etc that is used for large data processing, Data Analytics, and heavy scientific computing. If your system is not pre-equipped with Anaconda Navigator, you can learn **how to install Anaconda Navigator on Windows or Linux.**

Install and Run Pandas from Anaconda Navigator

Step 1: Search for **Anaconda Navigator** in Start Menu and open it.

Step 2: Click on the **Environment tab** and then click on the **Create** button to create a new Pandas Environment.

Step 3: Give a name to your Environment, e.g. Pandas, and then choose a Python and its version to run in the environment. Now click on the **Create** button to create Pandas Environment.

Step 4: Now click on the **Pandas Environment** created to activate it.

Activate the environment

Step 5: In the list above package names, select **All** to filter all the packages.

Getting all the packages

Step 6: Now in the Search Bar, look for '**Pandas**'. Select the **Pandas package** for Installation.

Step 7: Now Right Click on the checkbox given before the name of the package and then go to '**Mark for specific version installation**'. Now select the version that you want to install.

Step 8: Click on the **Apply** button to install the Pandas Package.

Step 9: Finish the Installation process by clicking on the **Apply** button.

Step 10: Now to open the Pandas Environment, click on the **Green Arrow** on the right of the package name and select the Console with which you want to begin your Pandas programming.

Pandas Terminal Window:

Install Pandas on Linux

Install Pandas on Linux, just type the following command in the Terminal Window and press Enter. Linux will automatically download and install the packages and files required to run Pandas Environment in Python:

```
pip3 install pandas
```

Install Pandas on MacOS

Install Pandas on MacOS, type the following command in the Terminal, and make sure that python is already installed in your system.

```
pip install pandas
```

How To Use Jupyter Notebook - An Ultimate Guide

The **Jupyter Notebook** is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. Jupyter has support for over 40 different programming languages and Python is one of them.

Installation of Jupyter Notebook

Using Anaconda

To install Jupyter Notebook using Anaconda: Download and install the latest Python 3 version of Anaconda. It includes Jupyter Notebook, Python, and other essential packages by default, making it an easy and recommended option for beginners.

Using pip

Alternatively you can install Jupyter Notebook using pip

python3 -m pip install --upgrade pip

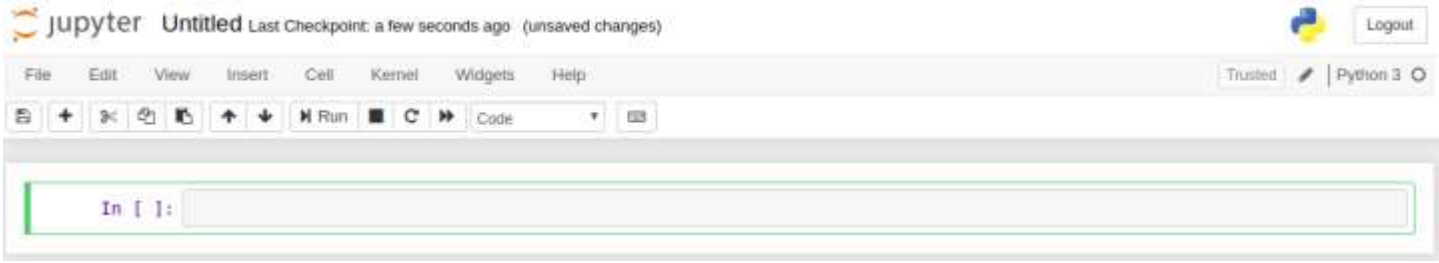
python3 -m pip install jupyter

Starting Jupyter Notebook

To launch Jupyter Notebook enter the following command in the terminal:

jupyter notebook

Creating a Notebook

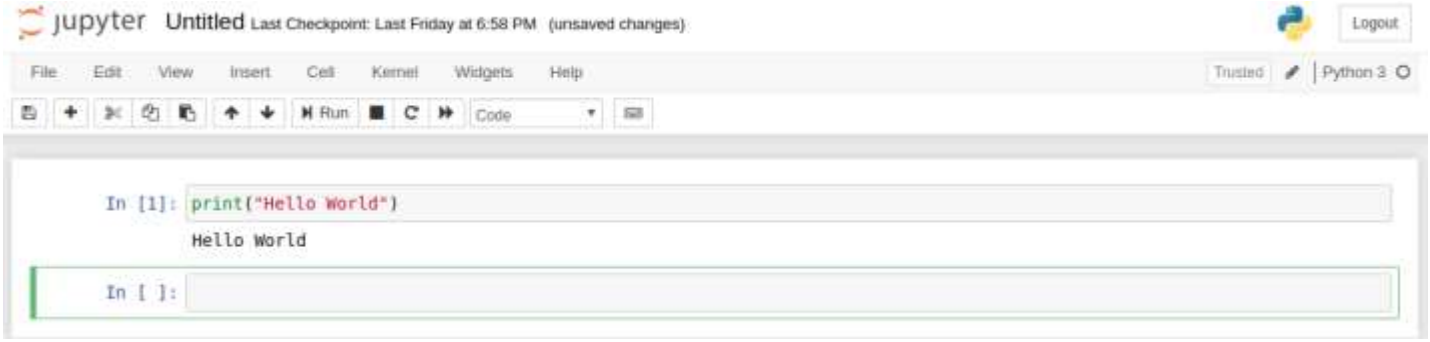


Writing and Running Code in Jupyter Notebook

After successfully installing and creating a notebook in Jupyter Notebook let's see how to write code in it. Jupyter notebooks consist of cells where you can write and execute code. For example, if you created a Python3 notebook then you can write Python3 code in the cell. Now, let's add the following code -

print("Hello, World!")

To run a cell either click the run button or press shift ⇧ + enter ⏎ after selecting the cell you want to execute. After writing the above code in the jupyter notebook the output was:



Cells in Jupyter Notebook

Cells can be considered as the body of the Jupyter. In the above screenshot the box with the green outline is a cell. There are 3 types of cell:

- **Code**
- **Markup**
- **Raw NBConverter**

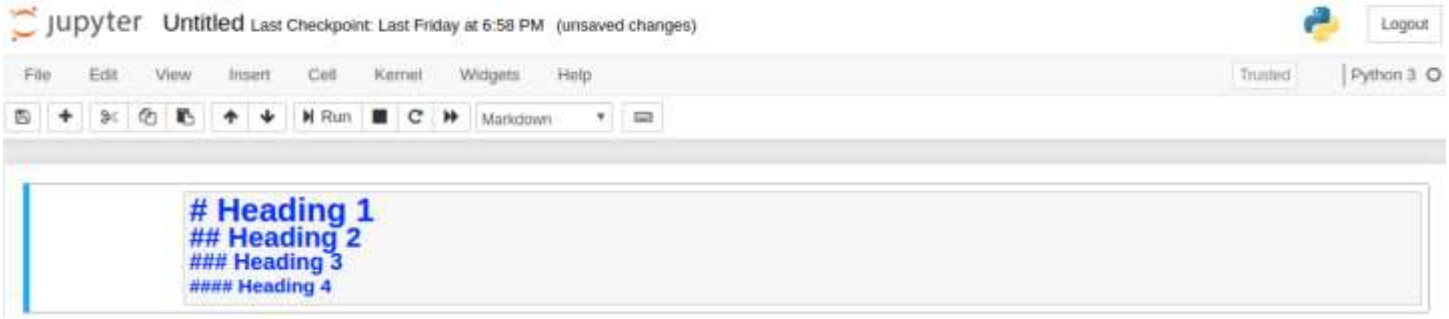
Code

This is where the code is typed and when executed the code will display the output below the cell. The type of code depends on the type of the notebook you have created. Consider the below example where a simple code of the Fibonacci series is created and this code also takes input from the user.

The text bar in the above code is prompted for taking input from the user. The output of the above code is as follows:

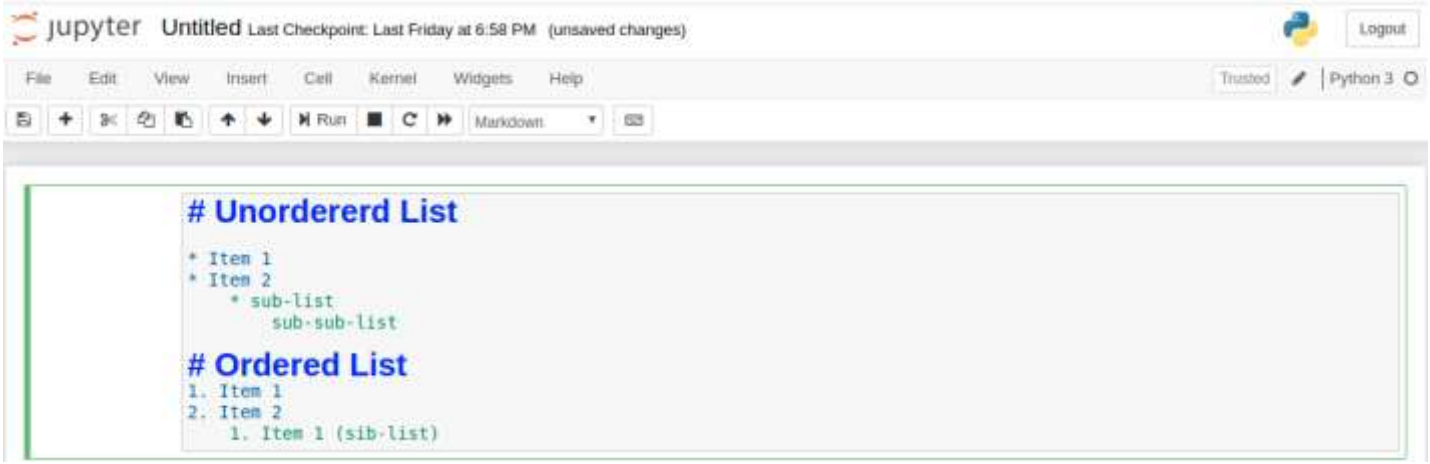
Output:

Adding Headers: Heading can be added by prefixing any line by single or multiple '#' followed by space.

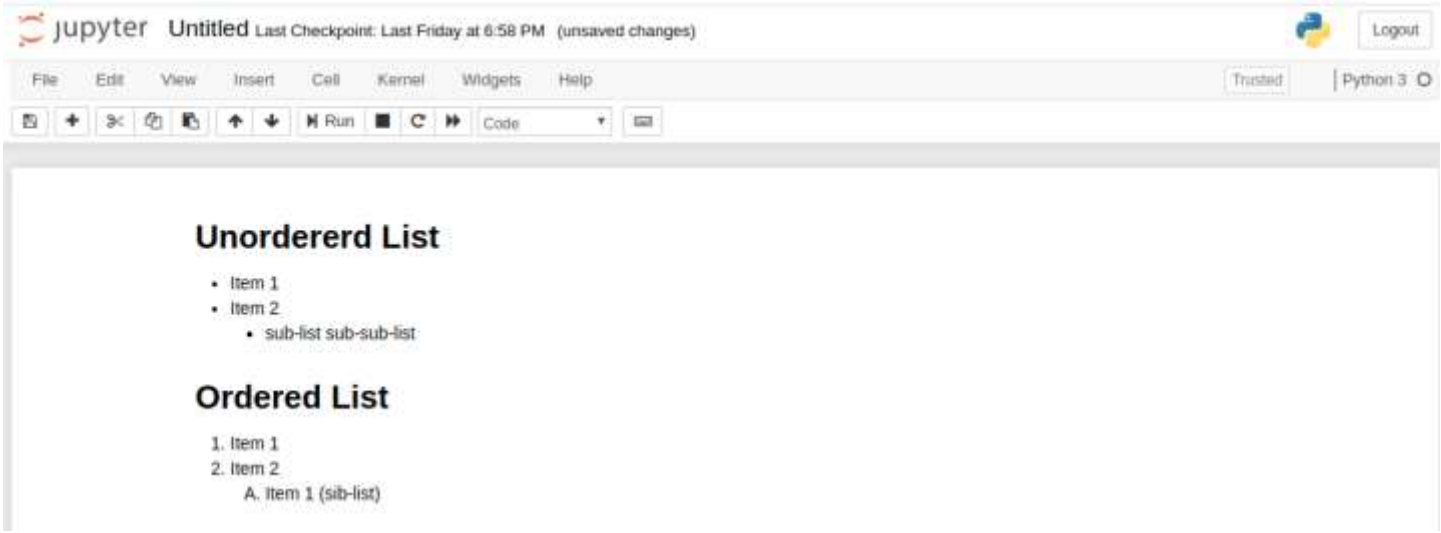


Output:

Adding List: Adding List is really simple in Jupyter Notebook. The list can be added by using '*' sign. And the Nested list can be created by using indentation. **Example:**



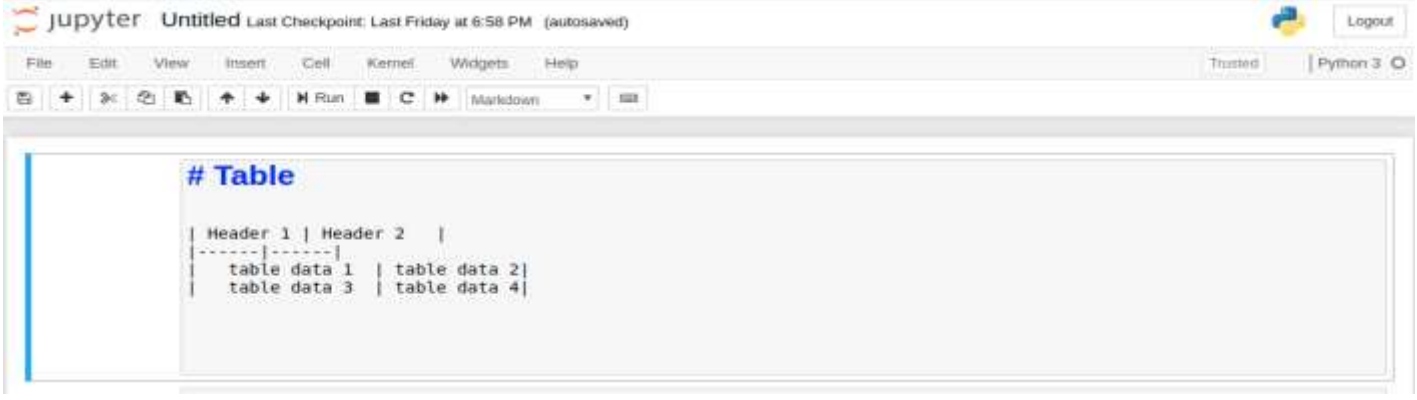
Output:



Adding Latex Equations: Latex expressions can be added by surrounding the latex code by '$' and for writing the expressions in the middle, surrounds the latex code by '$$'. **Example:**

Output:

Adding Table: A table can be added by writing the content in the following format.



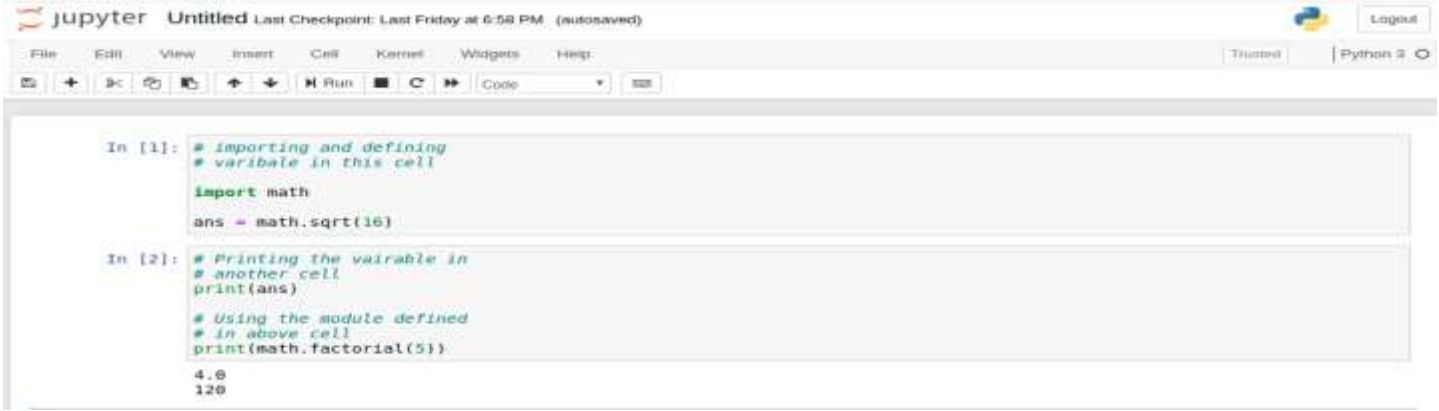
Output:

Raw NBConverter

Raw cells are provided to write the output directly. This cell is not evaluated by Jupyter notebook. After passing through nbconvert the raw cells arrives in the destination folder without any modification. For example one can write full Python into a raw cell that can only be rendered by Python only after conversion by nbconvert.

Understanding Jupyter Notebook Kernels

A **kernel** runs behind every Jupyter notebook, executing code and storing variables. The kernel remains active throughout the notebook session. For example, if a module is imported in one cell then that module will be available for the whole document. See the below example for better understanding.



Options for kernels: Jupyter Notebook provides various options for kernels. This can be useful if you want to reset things. The options are:

- **Restart:** This will restart the kernels i.e. clearing all the variables that were defined, clearing the modules that were imported, etc.
- **Restart and Clear Output:** This will do the same as above but will also clear all the output that was displayed below the cell.
- **Restart and Run All:** This is also the same as above but will also run all the cells in the top-down order.
- **Interrupt:** This option will interrupt the kernel execution. It can be useful in the case where the programs continue for execution or the kernel is stuck over some computation.

Naming and Saving Notebooks

By default a new notebook is named Untitled. To rename it: Click the notebook title and then enter a new name and confirm. Jupyter automatically saves notebooks periodically but you can manually save them by clicking File > Save and Checkpoint (Ctrl + S).

Extending Jupyter Notebook with Extensions

New functionality can be added to Jupyter through extensions. Extensions are javascript module. You can even write your own extension that can access the page's DOM and the Jupyter Javascript API. Jupyter supports four types of extensions.

- Kernel
- IPyhton Kernel
- Notebook
- Notebook server

Installing Extensions

Most of the extensions can be installed using Python's pip tool. If an extension can not be installed using pip then install the extension using the below command.
jupyter nbextension install extension_name
The above only installs the extension but does not enables it. To enable it type the below command in the terminal.
jupyter nbextension enable extension_name

Pandas DataFrame

A DataFrame is a two-dimensional, size-mutable and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

- Creating a DataFrame
- Pandas Dataframe Index
- Pandas Access DataFrame
- Indexing and Selecting Data with Pandas
- Slicing Pandas Dataframe
- Filter Pandas Dataframe with multiple conditions
- Merging, Joining and Concatenating Dataframes
- Sorting Pandas DataFrame
- Pivot Table in Pandas

Creating a Pandas DataFrame

Pandas DataFrame comes is a powerful tool that allows us to store and manipulate data in a structured way, similar to an Excel spreadsheet or a SQL table. A DataFrame is similar to a table with rows and columns. It helps in handling large amounts of data, performing calculations, filtering information with ease.

Creating an Empty DataFrame
An empty DataFrame in pandas is a table with no data but can have defined column names and indexes. It is useful for setting up a structure before adding data dynamically. An empty DataFrame can be created just by calling a dataframe constructor.

```
import pandas as pd

df = pd.DataFrame()

print(df)
```

Output

```
Empty DataFrame

Columns: []

Index: []
```

Creating a DataFrame from a List
A simple way to create a DataFrame is by using a single list. Pandas automatically assigns index values to the rows when you pass a list.

- Each item in the list becomes a row.
- The DataFrame consists of a single unnamed column.

```
import pandas as pd

lst = ['Tech', 'For', 'Tech', 'is',
```

```
        'portal', 'for', 'Tech']

df = pd.DataFrame(lst)
print(df)
```

Output

```
      0
0  Tech
1    For
2  Tech
3     is
4  portal
5    for
6  Tech
```

Creating DataFrame from dict of Numpy Array

We can create a Pandas DataFrame using a dictionary of NumPy arrays. Each key in the dictionary represents a column name and the corresponding NumPy array provides the values for that column.

```
import numpy as np
import pandas as pd

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
df = pd.DataFrame(data, columns=['A', 'B', 'C'])
print(df)
```

Output

```
  A B C
0 1 2 3
1 4 5 6
2 7 8 9
```

Creating a DataFrame from a List of Dictionaries

We can also create dataframe using List of Dictionaries. It represents data where each dictionary corresponds to a row. This method is useful for handling structured data from APIs or JSON files. It is commonly used in web scraping and API data processing since JSON responses often contain lists of dictionaries.

```
import pandas as pd

dict = {'name':["aparna", "pankaj", "sudhir", "Geeku"],
      'degree': ["MBA", "BCA", "M.Tech", "MBA"],
      'score':[90, 40, 80, 98]}

df = pd.DataFrame(dict)
print(df)
```

Output

```
    name  degree  score
0  aparna    MBA    90
1  pankaj    BCA    40
2  sudhir  M.Tech    80
3   Geeku    MBA    98
```

Pandas Dataframe Index

Index in pandas dataframe act as reference for each row in dataset. It can be numeric or based on specific column values. The default index is usually a **RangeIndex** starting from 0, but you can customize it for better data understanding. You can easily access the current index of a dataframe using the index attribute. **Let's us understand with the help of an example:**

1. Accessing and Modifying the Index

```
import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob', 'Eve', 'Charlie'],
    'Age': [25, 30, 22, 35, 28],
    'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
    'Salary': [50000, 55000, 40000, 70000, 48000]}

df = pd.DataFrame(data)
print(df.index)  # Accessing the index
```

Output

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

2. Setting a Custom Index

To set a custom index, you can use the set_index() method, allowing you to set a custom index based on a column, such as Name or Age.

```
# Set 'Name' column as the index
df_with_index = df.set_index('Name')
print(df_with_index)
```

Output

```
      Age  Gender  Salary
Name
John     25    Male   50000
Alice    30  Female   55000
Bob      22    Male   40000
Eve      35  Female   70000
Charlie  28    Male   480...
```

There are various operations you can perform with the DataFrame index, such as resetting it, changing it, or indexing with loc[]. Let's understand these as well:

3. Resetting the Index

If you need to reset the index back to **default integer index**, use reset_index() method. This will convert the **current index into a regular column and create a new default index.**

```
# Reset the index back to the default integer index
df_reset = df.reset_index()
print(df_reset)
```

Output

```
    Name  Age  Gender  Salary
0   John   25    Male   50000
1  Alice   30  Female   55000
2    Bob   22    Male   40000
```

```
3     Eve   35  Female   70000
4 Charlie   28    Male   48000
```

4. Indexing with loc

The loc[] method in pandas allows to access rows and columns of a dataFrame using their labels, making it easy to retrieve specific data points.

```
row = df.loc['Alice']
print(row)
```

Output

```
Age         30
Gender    Female
Salary     55000
Name: Alice, dtype: object
```

5. Changing the Index

Change the index of dataFrame, with help of set_index() method; allows to set one or more columns as the new index.

```
# Set 'Age' as the new index
df_with_new_index = df.set_index('Age')
print(df_with_new_index)
```

Output

```
      Name  Gender  Salary
Age
25     John    Male   50000
30    Alice  Female   55000
22      Bob    Male   40000
35      Eve  Female   70000
28  Charlie    Male   480...
```

Pandas Access DataFrame

Last Updated : 23 Jul, 2025

Accessing a dataframe in pandas involves retrieving, exploring, and manipulating **data stored within this structure.** The most basic form of accessing a DataFrame is simply referring to it by its variable name. This will display the entire DataFrame, which includes all rows and columns.

```
import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob', 'Eve', 'Charlie'],
        'Age': [25, 30, 22, 35, 28],
        'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
        'Salary': [50000, 55000, 40000, 70000, 48000]}

df = pd.DataFrame(data)
# Display the entire DataFrame
print(df)
```

Output

```
   Name  Age  Gender  Salary
0    John   25    Male   50000
1   Alice   30  Female   55000
2     Bob   22    Male   40000
3     Eve   35  Female   70000
4 Charlie   28    Male   48000
```

In addition to accessing the entire DataFrame there are several other methods to effectively retrieve and manipulate data within a Pandas DataFrame. Let's have a look on that:

1. Accessing Columns From DataFrame

Columns in a DataFrame can be accessed individually using bracket notation Accessing a column retrieves that column as a Series, which can then be further manipulated.

```
# Access the 'Age' column
age_column = df['Age']
print(age_column)
```

Output

```
0   25
1   30
2   22
3   35
4   28
Name: Age, dtype: int64
```

2. Accessing Rows by Index

To access specific rows in a DataFrame, you can use iloc (for positional indexing) or loc (for label-based indexing). These methods allow you to retrieve rows based on their index positions or labels.

```
# Access the row at index 1 (second row)
second_row = df.iloc[1]
print(second_row)
```

Output

```
Name      Alice
Age          30
Gender   Female
Salary    55000
Name: 1, dtype: object
```

3. Accessing Multiple Rows or Columns

You can access multiple rows or columns at once by passing a list of column names or index positions. This is useful when you need to select several columns or rows for further analysis.

```
# Access the first three rows and the 'Name' and 'Age' columns
subset = df.loc[0:2, ['Name', 'Age']]
print(subset)
```

Output

```
   Name  Age
0  John   25
```

```
1  Alice   30
2    Bob   22
```

4. Accessing Rows Based on Conditions

Pandas allows you to **filter rows** based on conditions, which can be very powerful for exploring subsets of data that meet specific criteria.

```
# Access rows where 'Age' is greater than 25
filtered_data = df[df['Age'] > 25]
print(filtered_data)
```

Output

```
    Name  Age  Gender  Salary
1   Alice   30  Female   55000
3     Eve   35  Female   70000
4 Charlie   28    Male   48000
```

5. Accessing Specific Cells with at and iat

If you need to access a specific cell, you can use the .at[] method for label-based indexing and the .iat[] method for integer position-based indexing. These are optimized for fast access to single values.

```
# Access the 'Salary' of the row with label 2
salary_at_index_2 = df.at[2, 'Salary']
print(salary_at_index_2)
```

Output

```
40000
```

Indexing and Selecting Data with Pandas

Indexing and selecting data helps us to efficiently retrieve specific rows, columns or subsets of data from a DataFrame. Whether we're filtering rows based on conditions, extracting particular columns or accessing data by labels or positions, mastering these techniques helps to work effectively with large datasets. In this article, we'll see various ways to index and select data in Pandas which shows us how to access the parts of our dataset.

1. Indexing Data using the [] Operator

The **[] operator** is the basic and frequently used method for indexing in Pandas. It allows us to select columns and filter rows based on conditions. This method can be used to select individual columns or multiple columns.

1. Selecting a Single Column

To select a single column, we simply refer the column name inside square brackets.

Here we will be using NBA dataset which you can download it from here.

```
import pandas as pd

data = pd.read_csv("/content/nba.csv", index_col="Name")
print("Dataset")
display(data.head(5))

first = data["Age"]
print("\nSingle Column selected from Dataset")
display(first.head(5))
```

Output:

Dataset

Name	Team	Number	Position	Age	Height	Weight	College	Salary
Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0

Single Column selected from Dataset

Name	Age
Avery Bradley	25.0
Jae Crowder	25.0
John Holland	27.0
R.J. Hunter	22.0
Jonas Jerebko	29.0

dtype: float64

Single column

2. Selecting Multiple Columns

To select multiple columns, pass a list of column names inside the [] operator:

```
first = data[["Age", "College", "Salary"]]
print("\nMultiple Columns selected from Dataset")
display(first.head(5))
```

Multiple Columns selected from Dataset

Name	Age	College	Salary
Avery Bradley	25.0	Texas	7730337.0
Jae Crowder	25.0	Marquette	6796117.0
John Holland	27.0	Boston University	NaN
R.J. Hunter	22.0	Georgia State	1148640.0
Jonas Jerebko	29.0	NaN	5000000.0

2. Indexing with .loc[ ]

The**.loc[]** function is used for label-based indexing. It allows us to access rows and columns by their labels. Unlike the indexing operator, it can select subsets of rows and columns simultaneously which offers flexibility in data retrieval.

1. Selecting a Single Row by Label

We can select a single row by its label:

```
import pandas as pd
data = pd.read_csv("/content/nba.csv", index_col="Name")

row = data.loc["Avery Bradley"]
```

print(row)

2. Selecting Multiple Rows by Label
To select multiple rows, pass a list of labels:
rows = data.loc[["Avery Bradley", "R.J. Hunter"]]
print(rows)
Output:

Name	Team	Number	Position	Age	Height	Weight	College	Salary
Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0

3. Selecting Specific Rows and Columns
We can select specific rows and columns by providing lists of row labels and column names:
Dataframe.loc[["row1", "row2"], ["column1", "column2", "column3"]]
selection = data.loc[["Avery Bradley", "R.J. Hunter"], ["Team", "Number", "Position"]]
print(selection)
Output:

Name	Team	Number	Position
Avery Bradley	Boston Celtics	0.0	PG
R.J. Hunter	Boston Celtics	28.0	SG

4. Selecting All Rows and Specific Columns
We can select all rows and specific columns by using a colon [:] to indicate all rows followed by the list of column names:
Dataframe.loc[:, ["column1", "column2", "column3"]]
all_rows_specific_columns = data.loc[:, ["Team", "Position", "Salary"]]

3. Indexing with .iloc[ ]
The **.iloc[]** function is used for position-based indexing. It allows us to access rows and columns by their integer positions. It is similar to **.loc[]** but only accepts integer-based indices to specify rows and columns.
1. Selecting a Single Row by Position
To select a single row using **.iloc[]** provide the integer position of the row:
import pandas as pd
data = pd.read_csv("/content/nba.csv", index_col="Name")
row = data.iloc[3]
print(row)
Output:

```
Team         Boston Celtics
Number                   28
Position                 SG
Age                      22
Height                  6-5
Weight                  185
College       Georgia State
Salary          1.14864e+06
Name: R.J. Hunter, dtype: object
```

2. Selecting Multiple Rows by Position
We can select multiple rows by passing a list of integer positions:
rows = data.iloc[[3, 5, 7]]
print(rows)

Output:

Name	Team	Number	Position	Age	Height	Weight	College	Salary
R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
Amir Johnson	Boston Celtics	90.0	PF	29.0	6-9	240.0	NaN	12000000.0
Kelly Olynyk	Boston Celtics	41.0	C	25.0	7-0	238.0	Gonzaga	2165160.0

3. Selecting Specific Rows and Columns by Position

We can select specific rows and columns by providing integer positions for both rows and columns:

```
selection = data.iloc[[3, 4], [1, 2]]
print(selection)
```

Output:

```
              Number Position
Name
R.J. Hunter     28.0       SG
Jonas Jerebko    8.0       PF
```

4. Selecting All Rows and Specific Columns by Position

To select all rows and specific columns, use a colon **[:]** for all rows and a list of column positions:

```
selection = data.iloc[:, [1, 2]]
print(selection)
```

4. Other Useful Indexing Methods

Pandas also provides several other methods that we may find useful for indexing and manipulating DataFrames:

1. **.head()**: Returns the first n rows of a DataFrame

```
print(data.head(5))
```

2. **.tail()**: Returns the last n rows of a DataFrame

```
print(data.tail(5))
```

3. **.at[]**: Access a single value for a row/column label pair

```
value = data.at["Avery Bradley", "Age"]
print(value)
```

Output:

25.0

4. **.query()**: Query the DataFrame using a boolean expression

```
result = data.query("Age > 25 and College == 'Duke'")
print(result)
```

More methods for indexing in a Pandas DataFrame include:

Function	Description
DataFrame.iat[]	Access a single value for a row/column pair by integer position.
DataFrame.pop()	Return item and drop from DataFrame.
DataFrame.xs()	Return a cross-section (row(s) or column(s)) from the DataFrame.
DataFrame.get()	Get item from object for given key (e.g DataFrame column).
DataFrame.isin()	Return a boolean DataFrame showing whether each element is contained in values.
DataFrame.where()	Return an object of the same shape with entries from self where cond is True otherwise from other.
DataFrame.mask()	Return an object of the same shape with entries from self where cond is False otherwise from other.
DataFrame.insert()	Insert a column into DataFrame at a specified location.

Slicing Pandas Dataframe

Slicing a Pandas DataFrame is a important skill for extracting specific data subsets. Whether you want to select rows, columns or individual cells, Pandas provides efficient methods like iloc[] and loc[]. In this guide we'll explore how to use integer-based and label-based indexing to slice DataFrames effectively.

Create a Custom Dataframe

Let's import pandas library and create pandas dataframe from custom nested list.

```
import pandas as pd

player_list = [['M.S.Dhoni', 36, 75, 5428000],
        ['A.B.D Villers', 38, 74, 3428000],
        ['V.Kohli', 31, 70, 8428000],
        ['S.Smith', 34, 80, 4428000],
        ['C.Gayle', 40, 100, 4528000],
        ['J.Root', 33, 72, 7028000],
        ['K.Peterson', 42, 85, 2528000]]

df = pd.DataFrame(player_list, columns=['Name', 'Age', 'Weight', 'Salary'])
print(df)
```

Output:

	Name	Age	Weight	Salary
0	M.S.Dhoni	36	75	5428000
1	A.B.D Villers	38	74	3428000
2	V.Kohli	31	70	8428000
3	S.Smith	34	80	4428000
4	C.Gayle	40	100	4528000
5	J.Root	33	72	7028000
6	K.Peterson	42	85	2528000

Slicing Using iloc[] (Integer-Based Indexing)

The iloc[] method in Pandas allows us to extract specific rows and columns based on their integer positions starting from 0. Each number represents a position in the DataFrame not the actual label of the row or column.

Slicing Rows in dataframe

Row slicing means selecting a specific set of rows from the DataFrame while keeping all columns.

```
df1 = df.iloc[0:4]
print(df1)
```

Output:

	Name	Age	Weight	Salary
0	M.S.Dhoni	36	75	5428000
1	A.B.D Villers	38	74	3428000
2	V.Kohli	31	70	8428000
3	S.Smith	34	80	4428000

Slicing Columns in dataframe

Column slicing means selecting a specific set of columns from the DataFrame while keeping all rows.

```
df1 = df.iloc[:, 0:2]

print(df1)
```

Output:

	Name	Age
0	M.S.Dhoni	36
1	A.B.D Villers	38
2	V.Kohli	31
3	S.Smith	34
4	C.Gayle	40
5	J.Root	33
6	K.Peterson	42

Selecting a Specific Cell in a Pandas DataFrame

If you need a single value from a DataFrame you can specify the exact row and column position

```
value = df.iloc[2, 3]
print("Specific Cell Value:", value)
```

Output:

Specific Cell Value: 8428000

Using Boolean Conditions in a Pandas DataFrame

Instead of selecting rows by index we can use Boolean conditions (e.g., Age > 35) to filter rows dynamically.

```
data = df[df['Age'] > 35].iloc[:, :]
print("\nFiltered Data based on Age > 35:\n", data)
```

Output:

```
Filtered Data based on Age > 35:
                Name  Age  Weight    Salary
0       M.S.Dhoni    36      75   5428000
1  A.B.D Villers     38      74   3428000
4          C.Gayle   40     100   4528000
6      K.Peterson    42      85   2528000
```

Slicing Using loc[]

We can also implement slicing using the loc function in Pandas but there are some limitations to be aware of. The loc function relies on labels meaning that if your DataFrame has custom labels instead of default integer indices you need to be careful with how you specify them.

Slicing Rows in Dataframe

With loc[] we can extract a range of rows by their labels instead of integer positions.

```
df.set_index('Name', inplace=True)

custom = df.loc['A.B.D Villers':'S.Smith']
print(custom)
```

Output:

```
                Age  Weight    Salary
Name
A.B.D Villers    38      74   3428000
V.Kohli          31      70   8428000
S.Smith          34      80   4428000
```

Selecting Specified cell in Dataframe

loc[] allows us to fetch a specific value based on row and column labels

```
value = df.loc['V.Kohli', 'Salary']
print("\nValue of the Specific Cell (V.Kohli, Salary):", value)
```

Output:

Value of the Specific Cell (V.Kohli, Salary): 8428000

Filter Pandas Dataframe with multiple conditions

The reason is dataframe may be having multiple columns and multiple rows. Selective display of columns with limited rows is always the expected view of users. To fulfill the user's expectations and also help in machine deep learning scenarios, filtering of Pandas dataframe with multiple conditions is much necessary.
Let us see the different ways to do the same.

Creating a sample dataframe to proceed further

```
# import module
import pandas as pd

# assign data
dataFrame = pd.DataFrame({'Name': [' RACHEL  ', ' MONICA  ', ' PHOEBE  ',
                    '  ROSS    ', 'CHANDLER', ' JOEY    '],

                'Age': [30, 35, 37, 33, 34, 30],

                'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

                'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',
                     'IT', 'ARTIST']})

# display dataframe
display(dataFrame)
```

Output:

	Name	Age	Salary	JOB
0	RACHEL	30	100000	DESIGNER
1	MONICA	35	93000	CHEF
2	PHOEBE	37	88000	MASUS
3	ROSS	33	120000	PALENTOLOGY
4	CHANDLER	34	94000	IT
5	JOEY	30	95000	ARTIST

Filter Pandas Dataframe with multiple conditions Using loc

Here we will get all rows having Salary greater or equal to 100000 and Age < 40 and their JOB starts with 'D' from the dataframe. Print the details with Name and their JOB. For the above requirement, we can achieve this by using **loc**. It is used to access single or more rows and columns by label(s) or by a boolean array. loc works with column labels and indexes.

```
# import module
import pandas as pd

# assign data
dataFrame = pd.DataFrame({'Name': [' RACHEL  ', ' MONICA  ', ' PHOEBE  ',
                    '  ROSS    ', 'CHANDLER', ' JOEY    '],

                'Age': [30, 35, 37, 33, 34, 30],

                'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

                'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',
                     'IT', 'ARTIST']})
# filter dataframe
display(dataFrame.loc[(dataFrame['Salary']>=100000) & (dataFrame['Age']< 40) &
(dataFrame['JOB'].str.startswith('D')),
          ['Name','JOB']])
```

Output:

	Name	JOB
0	RACHEL	DESIGNER

Output resolves for the given conditions and finally, we are going to show only 2 columns namely Name and JOB.

Filter Pandas Dataframe Using NumPy

Here will get all rows having Salary greater or equal to 100000 and Age < 40 and their JOB starts with 'D' from the data frame. We need to use NumPy.

```
# import module
import pandas as pd
import numpy as np

# assign data
dataFrame = pd.DataFrame({'Name': [' RACHEL  ', ' MONICA  ', ' PHOEBE  ',
                    '  ROSS   ', 'CHANDLER', ' JOEY   '],

                'Age': [30, 35, 37, 33, 34, 30],

                'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

                'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',
                     'IT', 'ARTIST']})

# filter dataframe
filtered_values = np.where((dataFrame['Salary']>=100000) & (dataFrame['Age']< 40) &
(dataFrame['JOB'].str.startswith('D')))
print(filtered_values)
display(dataFrame.loc[filtered_values])
```

Output:

```
(array([0], dtype=int64),)
```

	Name	Age	Salary	JOB
0	RACHEL	30	100000	DESIGNER

In the above example, *print(filtered_values)* will give the output as *(array([0], dtype=int64),)* which indicates the first row with index value 0 will be the output. After that output will have 1 row with all the columns and it is retrieved as per the given conditions.

Filter Pandas Dataframe Using Query (eval and query works only with columns)

In this approach, we get all rows having Salary lesser or equal to 100000 and Age < 40, and their JOB starts with 'C' from the dataframe. Its just query the columns of a DataFrame with a single or more Boolean expressions and if multiple, it is having & condition in the middle.

```
# import module
import pandas as pd

# assign data
dataFrame = pd.DataFrame({'Name': [' RACHEL  ', ' MONICA  ', ' PHOEBE  ',
                    '  ROSS   ', 'CHANDLER', ' JOEY   '],

                'Age': [30, 35, 37, 33, 34, 30],
```

```
                'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

                'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',
                    'IT', 'ARTIST']})

# filter dataframe
display(dataFrame.query('Salary  <= 100000 & Age < 40 & JOB.str.startswith("C").values'))
```

Output:

	Name	Age	Salary	JOB
1	MONICA	35	93000	CHEF

Pandas Boolean indexing multiple conditions standard way ("Boolean indexing" works with values in a column only)

In this approach, we get all rows having Salary lesser or equal to 100000 and Age < 40 and their JOB starts with 'P' from the dataframe. In order to select the subset of data using the values in the dataframe and applying Boolean conditions, we need to follow these ways

```
# import module
import pandas as pd

# assign data
dataFrame = pd.DataFrame({'Name': [' RACHEL  ', ' MONICA  ', ' PHOEBE  ',
                    '  ROSS   ', 'CHANDLER', ' JOEY   '],

                'Age': [30, 35, 37, 33, 34, 30],

                'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

                'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',
                    'IT', 'ARTIST']})

# filter dataframe
display(dataFrame[(dataFrame['Salary']>=100000)  &  (dataFrame['Age']<40)  &
dataFrame['JOB'].str.startswith('P')][['Name','Age','Salary']])
```

Output:

	Name	Age	Salary
3	ROSS	33	120000

We are mentioning a list of columns that need to be retrieved along with the Boolean conditions and since many conditions, it is having '&'.

Eval multiple conditions ("eval" and "query" works only with columns)

Here, we get all rows having Salary lesser or equal to 100000 and Age < 40 and their JOB starts with 'A' from the dataframe.

```
# import module
import pandas as pd

# assign data
dataFrame = pd.DataFrame({'Name': [' RACHEL  ', ' MONICA  ', ' PHOEBE  ',
                    '  ROSS   ', 'CHANDLER', ' JOEY   '],

                'Age': [30, 35, 37, 33, 34, 30],

                'Salary': [100000, 93000, 88000, 120000, 94000, 95000],
```

```
            'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',
                'IT', 'ARTIST']})

# filter dataframe
display(dataFrame[dataFrame.eval("Salary <=100000 & (Age <40) & JOB.str.startswith('A').values")])
```

Output:

	Name	Age	Salary	JOB
5	JOEY	30	95000	ARTIST

Python | Pandas Merging, Joining and Concatenating

Concatenating DataFrames

Concatenating DataFrames means combining them either by stacking them on top of each other (vertically) or placing them side by side (horizontally). In order to Concatenate dataframe, we use different methods which are as follows:

1. Concatenating DataFrame using .concat()

To concatenate DataFrames, we use the **pd.concat()** function. This function allows us to combine multiple DataFrames into one by specifying the axis (rows or columns).

Here we will be loading and printing the custom dataset, then we will perform the concatenation using pd.concat().

```
import pandas as pd

data1 = {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],
      'Age': [27, 24, 22, 32],
      'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
      'Qualification': ['Msc', 'MA', 'MCA', 'Phd']}

data2 = {'Name': ['Abhi', 'Ayushi', 'Dhiraj', 'Hitesh'],
      'Age': [17, 14, 12, 52],
      'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
      'Qualification': ['Btech', 'B.A', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1, index=[0, 1, 2, 3])

df1 = pd.DataFrame(data2, index=[4, 5, 6, 7])

print(df, "\n\n", df1)
```

Now we apply .concat function in order to concat two dataframe.

```
frames = [df, df1]

res1 = pd.concat(frames)
res1
```

2. Concatenating DataFrames by Setting Logic on Axes

We can modify the concatenation by setting logic on the axes. Specifically we can choose whether to take the Union (join='outer') or Intersection (join='inner') of columns.

```
import pandas as pd

data1 = {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],
      'Age': [27, 24, 22, 32],
      'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
      'Qualification': ['Msc', 'MA', 'MCA', 'Phd'],
```

```
    'Mobile No': [97, 91, 58, 76]}

data2 = {'Name': ['Gaurav', 'Anuj', 'Dhiraj', 'Hitesh'],
    'Age': [22, 32, 12, 52],
    'Address': ['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
    'Qualification': ['MCA', 'Phd', 'Bcom', 'B.hons'],
    'Salary': [1000, 2000, 3000, 4000]}

df = pd.DataFrame(data1, index=[0, 1, 2, 3])

df1 = pd.DataFrame(data2, index=[2, 3, 6, 7])

print(df, "\n\n", df1)
Now we set axes join = inner for intersection of dataframe which keeps only the common columns.
res2 = pd.concat([df, df1], axis=1, join='inner')

res2
Now we set axes join = outer for union of dataframe which keeps all columns from both DataFrames.
res2 = pd.concat([df, df1], axis=1, sort=False)

res2
```

3. Concatenating DataFrames by Ignoring Indexes

Sometimes the indexes of the original DataFrames may not be relevant. We can ignore the indexes and reset them using the **ignore_index** argument. This is useful when we don't want to carry over any index information.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age':[27, 24, 22, 32],
    'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
    'Qualification':['Msc', 'MA', 'MCA', 'Phd'],
    'Mobile No': [97, 91, 58, 76]}

data2 = {'Name':['Gaurav', 'Anuj', 'Dhiraj', 'Hitesh'],
    'Age':[22, 32, 12, 52],
    'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
    'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons'],
    'Salary':[1000, 2000, 3000, 4000]}

df = pd.DataFrame(data1,index=[0, 1, 2, 3])

df1 = pd.DataFrame(data2, index=[2, 3, 6, 7])


print(df, "\n\n", df1)

Now we are going to apply ignore_index as an argument.
res = pd.concat([df, df1], ignore_index=True)

res
```

4. Concatenating DataFrame with group keys :

If we want to retain information about the DataFrame from which each row came, we can use the **keys argument**. This assigns a label to each group of rows based on the source DataFrame.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
```

```
    'Age':[27, 24, 22, 32],
    'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
    'Qualification':['Msc', 'MA', 'MCA', 'Phd']}

data2 = {'Name':['Abhi', 'Ayushi', 'Dhiraj', 'Hitesh'],
    'Age':[17, 14, 12, 52],
    'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
    'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1,index=[0, 1, 2, 3])

df1 = pd.DataFrame(data2, index=[4, 5, 6, 7])

print(df, "\n\n", df1)
```

Here we will use keys as an argument. The **keys** argument creates a hierarchical index where each row is labeled with the source DataFrame (df1 or df2).

```
frames = [df, df1 ]

res = pd.concat(frames, keys=['x', 'y'])
res
```

5. Concatenating Mixed DataFrames and Series

We can also concatenate a mix of Series and DataFrames. If we include a Series in the list, it will automatically be converted to a DataFrame and we can specify the column name.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age':[27, 24, 22, 32],
    'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
    'Qualification':['Msc', 'MA', 'MCA', 'Phd']}

df = pd.DataFrame(data1,index=[0, 1, 2, 3])

s1 = pd.Series([1000, 2000, 3000, 4000], name='Salary')

print(df, "\n\n", s1)
```

Merging DataFrame

Merging DataFrames in Pandas is similar to performing SQL joins. It is useful when we need to combine two DataFrames based on a common column or index. The **merge()** function provides flexibility for different types of joins.

There are four basic ways to handle the join (inner, left, right and outer) depending on which rows must retain their data.

1. Merging DataFrames Using One Key

We can merge DataFrames based on a common column by using the on argument. This allows us to combine the DataFrames where values in a specific column match.

```
import pandas as pd

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
     'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age':[27, 24, 22, 32],}

data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
     'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
    'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}
```

```
df = pd.DataFrame(data1)

df1 = pd.DataFrame(data2)


print(df, "\n\n", df1)
```

2. Merging DataFrames Using Multiple Keys

We can also merge DataFrames based on more than one column by passing a list of column names to the on argument.

```
import pandas as pd

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
      'key1': ['K0', 'K1', 'K0', 'K1'],
      'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
     'Age':[27, 24, 22, 32],}

data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
      'key1': ['K0', 'K0', 'K0', 'K0'],
      'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
     'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1)

df1 = pd.DataFrame(data2)


print(df, "\n\n", df1)
```

Now we merge dataframe using multiple keys.

```
res1 = pd.merge(df, df1, on=['key', 'key1'])

res1
```

3. Merging DataFrames Using the how Argument

We use **how** argument to merge specifies how to find which keys are to be included in the resulting table. If a key combination does not appear in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names:

MERGE METHOD	JOIN NAME	DESCRIPTION
left	LEFT OUTER JOIN	Use keys from left frame only
right	RIGHT OUTER JOIN	Use keys from right frame only
outer	FULL OUTER JOIN	Use union of keys from both frames
inner	INNER JOIN	Use intersection of keys from both frames

```
import pandas as pd

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
      'key1': ['K0', 'K1', 'K0', 'K1'],
      'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
     'Age':[27, 24, 22, 32],}

data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
      'key1': ['K0', 'K0', 'K0', 'K0'],
      'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
     'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}
```

```
df = pd.DataFrame(data1)

df1 = pd.DataFrame(data2)


print(df, "\n\n", df1)
```

from left frame only. In this it includes all rows from the left DataFrame and only matching rows from the right.

```
res = pd.merge(df, df1, how='left', on=['key', 'key1'])

res
```

Now we set how = 'right' in order to use keys from right frame only. In this it includes all rows from the right DataFrame and only matching rows from the left.

```
res1 = pd.merge(df, df1, how='right', on=['key', 'key1'])

res1
```

Now we set how = 'outer' in order to get **union** of keys from dataframes. In this it combines all rows from both DataFrames, filling missing values with NaN.

```
res2 = pd.merge(df, df1, how='outer', on=['key', 'key1'])

res2
```

Now we set how = 'inner' in order to get intersection of keys from dataframes. In this it only includes rows where there is a match in both DataFrames.

```
res3 = pd.merge(df, df1, how='inner', on=['key', 'key1'])

res3
```

Joining DataFrame

The **.join()** method in Pandas is used to combine columns of two DataFrames based on their indexes. It's a simple way of merging two DataFrames when the relationship between them is primarily based on their row indexes. It is used when we want to combine DataFrames along their indexes rather than specific columns.

1. Joining DataFrames Using .join()

If both DataFrames have the same index, we can use the **.join()** function to combine their columns. This method is useful when we want to merge DataFrames based on their row indexes rather than columns.

```
import pandas as pd


data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age':[27, 24, 22, 32]}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
    'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])


print(df, "\n\n", df1)
```

Now we are using .join() method in order to join dataframes

```
res = df.join(df1)

res
```

Now we use how = 'outer' in order to get union

```
res1 = df.join(df1, how='outer')

res1
```

2. Joining DataFrames Using the "on" Argument

If we want to join DataFrames based on a column (rather than the index), we can use the on argument. This allows us to specify which column(s) should be used to align the two DataFrames.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age':[27, 24, 22, 32],
    'Key':['K0', 'K1', 'K2', 'K3']}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
    'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1)

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])


print(df, "\n\n", df1)
Now we are using .join with “on” argument.
res2 = df.join(df1, on='Key')

res2
```

3. Joining DataFrames with Different Index Levels (Multi-Index)

In some cases, we may be working with DataFrames that have multi-level indexes. The **.join()** function also supports joining DataFrames that have different index levels by specifying the index levels.

```
import pandas as pd

data1 = {'Name':['Jai', 'Princi', 'Gaurav'],
    'Age':[27, 24, 22]}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kanpur'],
    'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1, index=pd.Index(['K0', 'K1', 'K2'], name='key'))

index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
                        ('K2', 'Y2'), ('K2', 'Y3')],
                        names=['key', 'Y'])

df1 = pd.DataFrame(data2, index= index)


print(df, "\n\n", df1)
Now we join singly indexed dataframe with multi-indexed dataframe.
result = df.join(df1, how='inner')

result
```

How to Sort Pandas DataFrame

Sorting data is an important step in data analysis as it helps to organize and structure the information for easier interpretation and decision-making. Whether we're working with small datasets or large ones, sorting allows us to arrange data in a meaningful way.

Pandas provides the **sort_values()** method which allows us to sort a DataFrame by one or more columns in either ascending or descending order.

1. Sorting a DataFrame by a Single Column

The **sort_values()** method in Pandas makes it easy to sort our DataFrame by a single column. By default, it sorts in ascending order but we can customize this.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'Score': [85, 90, 95, 80]}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age')
print(sorted_df)
```

Output:

	Name	Age	Score
0	Alice	25	85
1	Bob	30	90
2	Charlie	35	95
3	David	40	80

In this example, the DataFrame is sorted by the Age column in ascending order but here it is already sorted. If we need the sorting to be in **descending** order simply pass **ascending=False:**

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],'Age': [25, 30, 35, 40],'Score': [85, 90, 95, 80]}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age',ascending=False)
print(sorted_df)
```

Parameters of sort_values():

- **by**: Specifies the column to sort by.
- **ascending**: A boolean (True for ascending, False for descending).
- **inplace**: If True, the original DataFrame is modified otherwise a new sorted DataFrame is returned.
- **na_position**: Controls where NaN values are placed. Use 'first' to put NaNs at the top or 'last' (default) to place them at the end.
- **ignore_index**: If True, resets the index after sorting.

2. Sorting a DataFrame by Multiple Columns

When sorting by multiple columns, Pandas allows us to specify a list of column names. This is useful when we want to sort by one column like age and if there are ties, sort by another column like salary.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'Score': [85, 90, 95, 80]}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by=['Age', 'Score'])
print(sorted_df)
```

This will sort first by Age and if multiple rows have the same Age, it will then sort those rows by Salary.

3. Sorting DataFrame with Missing Values

In real-world datasets, missing values (NaNs) are common. By default sort_values() places NaN values at the end. If we need them at the top, we can use the **na_position** parameter.

```
import pandas as pd
data_with_nan = {"Name": ["Alice", "Bob", "Charlie", "David"],"Age": [28, 22, None, 22]}
df_nan = pd.DataFrame(data_with_nan)
```

```
sorted_df = df_nan.sort_values(by="Age", na_position="first")
print(sorted_df)
```

4. Sorting by Index

In addition to sorting by column values, we may also want to sort a DataFrame based on its **index**. This can be done using the **sort_index()** method in Pandas. By default, **sort_index**() sorts the DataFrame based on the index in **ascending order.**

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
     'Age': [25, 30, 35, 40],
     'Score': [85, 90, 95, 80]}
df = pd.DataFrame(data)

df_sorted_by_index = df.sort_index()
print(df_sorted_by_index)
```

We can also sort by index in **descending** order by passing the **ascending=False** argument.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
     'Age': [25, 30, 35, 40],
     'Score': [85, 90, 95, 80]}
df = pd.DataFrame(data)
df_sorted_by_index_desc = df.sort_index(ascending=False)
print(df_sorted_by_index_desc)
```

5. Choosing a Sorting Algorithm

Pandas provides different sorting algorithms that we can choose using the **kind** parameter. Available options are:

1. QuickSort (kind='quicksort'): It is a highly efficient, divide-and-conquer sorting algorithm. It selects a "pivot" element and partitions the dataset into two halves: one with elements smaller than the pivot and the other with elements greater than the pivot.

```
import pandas as pd

data = {
   "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
   "Age": [28, 22, 25, 22, 28],
   "Score": [85, 90, 95, 80, 88]
}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age', kind='quicksort')
print(sorted_df)
```

2. MergeSort (kind='mergesort'): Divides the dataset into smaller subarrays, sorts them and then merges them back together in sorted order.

```
import pandas as pd

data = {
   "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
   "Age": [28, 22, 25, 22, 28],
   "Score": [85, 90, 95, 80, 88]
}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age', kind='mergesort')
print(sorted_df)
```

3. HeapSort (kind= 'heapsort'): It is another comparison-based sorting algorithm that builds a heap data structure to systematically extract the largest or smallest element and reorder the dataset.

```
import pandas as pd

data = {
   "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
   "Age": [28, 22, 25, 22, 28],
   "Score": [85, 90, 95, 80, 88]
}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age', kind='heapsort')
print(sorted_df)
```

***Note**: HeapSort being unstable, may not preserve this order and in some cases like the one above, it swaps rows with the same sorting key.*

6. Applying Custom Sorting Logic

We can also apply custom sorting logic using the **key** parameter. This is useful when we need to sort strings in a specific way such as ignoring case sensitivity.

```
import pandas as pd
data = {
   "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
   "Age": [28, 22, 25, 22, 28],
   "Score": [85, 90, 95, 80, 88]
}
df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Name', key=lambda col: col.str.lower())
print(sorted_df)
```

How to Create a Pivot Table in Python using Pandas

A pivot table is a statistical table that summarizes a substantial table like a big dataset. It is part of data processing. This summary in pivot tables may include mean, median, sum, or other statistical terms. Pivot tables are originally associated with MS Excel but we can create a pivot table in Pandas using Python using the Pandas Dataframe pivot_table() method.

Creating a Sample DataFrame

Let's first create a dataframe that includes Sales of Fruits.

```
# importing pandas
import pandas as pd

# creating dataframe
df = pd.DataFrame({'Product': ['Carrots', 'Broccoli', 'Banana', 'Banana',
                               'Beans', 'Orange', 'Broccoli', 'Banana'],
                   'Category': ['Vegetable', 'Vegetable', 'Fruit', 'Fruit',
                                'Vegetable', 'Fruit', 'Vegetable', 'Fruit'],
                   'Quantity': [8, 5, 3, 4, 5, 9, 11, 8],
                   'Amount': [270, 239, 617, 384, 626, 610, 62, 90]})
df
```

Output

	Product	Category	Quantity	Amount
0	Carrots	Vegetable	8	270
1	Broccoli	Vegetable	5	239
2	Banana	Fruit	3	617
3	Banana	Fruit	4	384
4	Beans	Vegetable	5	626
5	Orange	Fruit	9	610
6	Broccoli	Vegetable	11	62
7	Banana	Fruit	8	90

Create a Pivot Table in Pandas

Below are some examples to understand how we can create a pivot table in Pandas in Python:

Example 1: Get the Total Sales of Each Product

In this example, the DataFrame 'df' is transformed using a pivot table, aggregating the total 'Amount' for each unique 'Product' and displaying the result with the sum of amounts for each product.

```
pivot = df.pivot_table(index=['Product'],
            values=['Amount'],
            aggfunc='sum')
print(pivot)
```

Output

```
          Amount
Product
Banana      1091
Beans        626
Broccoli     301
Carrots      270
Orange       610
```

Example 2: Get the Total Sales of Each Category

In this example, a pivot table is created from the DataFrame 'df' to summarize the total 'Amount' sales for each unique 'Category,' employing the 'sum' aggregation function, and the result is printed.

```
# creating pivot table of total
# sales category-wise aggfunc = 'sum'
pivot = df.pivot_table(index=['Category'],
            values=['Amount'],
            aggfunc='sum')
print(pivot)
```

Output

```
            Amount
Category
Fruit         1701
Vegetable     1197
```

Example 3: Get Total Sales by Category and Product Both

In this example, a pivot table is generated from the DataFrame 'df' to showcase the total 'Amount' sales for unique combinations of 'Product' and 'Category,' utilizing the 'sum' aggregation function. The resulting pivot table is then printed.

```
pivot = df.pivot_table(index=['Product', 'Category'],
            values=['Amount'], aggfunc='sum')
print(pivot)
```

Output

		Amount
Product	Category	
Banana	Fruit	1091
Beans	Vegetable	626
Broccoli	Vegetable	301
Carrots	Vegetable	270
Orange	Fruit	610

Example 4: Get the Mean, Median, Minimum Sale by Category

In this example, a pivot table is created from the DataFrame 'df' to display the median, mean, and minimum 'Amount' values categorized by 'Category.' The aggregation functions 'median,' 'mean,' and 'min' are applied, and the resulting pivot table is printed.

```
# 'mean', 'min'} will get median, mean and
# minimum of sales respectively
pivot = df.pivot_table(index=['Category'], values=['Amount'],
            aggfunc={'median', 'mean', 'min'})
print(pivot)
```

Output

	Amount		
	mean	median	min
Category			
Fruit	425.25	497.0	90.0
Vegetable	299.25	254.5	62.0

Pandas Series

A Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating-point numbers, Python objects, etc.). It's similar to a column in a spreadsheet or a database table.

- Creating a Series
- Accessing elements of a Pandas Series
- Binary Operations on Series
- Pandas Series Index() Methods
- Create a Pandas Series from array

Creating a Pandas Series

A **Pandas Series** is like a single column of data in a spreadsheet. ***It is a one-dimensional array that can hold many types of data such as numbers, words or even other Python objects.*** Each value in a Series is associated with an **index**, which makes data retrieval and manipulation easy. This article explores multiple ways to create a Pandas Series with step-by-step explanations and examples.

Creating an Empty Pandas Series

An empty Series contains no data and can be useful when we plan to add values later. we can create an empty Series using the pd.Series() function. By default an empty Series has a float64 data type. If we need a different data type specify it using the dtype parameter

```
import pandas as pd

ser = pd.Series()

print(ser)
```

Output:

Series([], dtype: float64)

Creating a Series from a NumPy Array

If we already have data stored in a **NumPy array** we can easily convert it into a Pandas Series. This is helpful when working with numerical data.

```
import pandas as pd
import numpy as np

data = np.array(['g', 'e', 'e', 'k', 's'])

ser = pd.Series(data)
print(ser)
```

Output:

```
0    g
1    e
2    e
3    k
4    s
dtype: object
```

Series Using Numpy Arrays

Creating a Series from a List

we can create a Series by passing a Python **list** to the pd.Series() function. Pandas automatically assigns an index to each element starting from 0. This is a simple way to store and manipulate data.

```
import pandas as pd

data_list = ['g', 'e', 'e', 'k', 's']

ser = pd.Series(data_list)
print(ser)
```

Output:

```
0    g
1    e
2    e
3    k
4    s
dtype: object
```

Creating a Series from a Dictionary

A dictionary in Python stores data as key-value pairs. When we convert Dictionary into a Pandas Series the keys become index labels and the values become the data. This method is useful for labeled data preserving structure and enabling quick access. Below is an example.

```
import pandas as pd

data_dict = {'Tech': 10, 'for': 20, 'Tech': 30}

ser = pd.Series(data_dict)
print(ser)
```

Creating a Series Using NumPy Functions

In order to create a series using numpy function. Some commonly used NumPy functions for generating sequences include numpy.linspace() for creating evenly spaced numbers over a specified range and numpy.random.randn() for generating random numbers from a normal distribution. This is particularly useful when working with scientific computations, statistical modeling or large datasets

```
import numpy as np
import pandas as pd

ser = pd.Series(np.linspace(1, 10, 5))
print(ser)
```

Output:

```
0     1.00
1     3.25
2     5.50
3     7.75
4    10.00
dtype: float64
```

Series using Numpy Functions

Creating a Series Using range()

The range() function in Python is commonly used to generate sequences of numbers and it can be easily converted into a Pandas Series. This is particularly useful for creating a sequence of values in a structured format without need of manually specify each element. Below is an how range() can be used to create a Series.

```
import pandas as pd

ser = pd.Series(range(5, 15))
print(ser)
```

Output:

```
0     5
1     6
2     7
3     8
4     9
5    10
6    11
7    12
8    13
9    14
dtype: int64
```

Series using range()

Creating a Series Using List Comprehension

List comprehension is a concise way to generate sequences and apply transformations in a single line of code. This method is useful when we need to create structured sequences dynamically. Below is an example demonstrating how list comprehension is used to create a Series with a custom index.

```
import pandas as pd
ser=pd.Series(range(1,20,3), index=[x for x in 'abcdefg'])
print(ser)
```

Output:

```
a     1
b     4
c     7
d    10
e    13
f    16
g    19
dtype: int64
```

Accessing elements of a Pandas Series

Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). Labels need not be unique but must be a hashable type. An element in the series can be accessed similarly to that in an **ndarray**. Elements of a series can be accessed in two ways:

- **Accessing Element from Series with Position**
- **Accessing Element Using Label (index)**

Accessing Element from Series with Position

In order to access the series element refers to the index number. Use the index operator **[ ]** to access an element in a series. The index must be an integer.

In order to access multiple elements from a series, we use Slice operation. Slice operation is performed on Series with the use of the colon(**:**). To print elements from beginning to a range use **[:Index]**, to print elements

from end-use **[:-Index]**, to print elements from specific Index till the end use **[Index:]**, to print elements within a range, use [Start Index:End Index] and to print whole Series with the use of slicing operation, use **[:]**. Further, to print the whole Series in reverse order, use **[::-**1].

Accessing the First Element of Series

In this example, a Pandas Series named 'ser' is created from a NumPy array 'data' containing the elements 'g', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'g', 'e', 'e', 'k', 's'. The first element of the series is accessed and printed using `print(ser[0])`

```
# import pandas and numpy
import pandas as pd
import numpy as np

# creating simple array
data = np.array(['g', 'e', 'e', 'k', 's', 'f',
                 'o', 'r', 'g', 'e', 'e', 'k', 's'])
ser = pd.Series(data)
# retrieve the first element
print(ser[0])
```

Output:

```
g
```

Accessing First 5 Elements of Series

In this example, a Pandas Series named 'ser' is created from a NumPy array 'data' containing the elements 'g', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'g', 'e', 'e', 'k', 's'. The first five elements of the series are accessed and printed using print(ser[:5]).

```
# import pandas and numpy
import pandas as pd
import numpy as np

# creating simple array
data = np.array(['g', 'e', 'e', 'k', 's', 'f',
                 'o', 'r', 'g', 'e', 'e', 'k', 's'])
ser = pd.Series(data)
# retrieve the first element
print(ser[:5])
```

Output:

```
0    g
1    e
2    e
3    k
4    s
dtype: object
```

Accessing Last 10 Elements of Series

In this example, a Pandas Series named 'ser' is created from a NumPy array 'data' containing the elements 'g', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'g', 'e', 'e', 'k', 's'. The last 10 elements of the series are accessed and printed using `print(ser[-10:])`.

```
# import pandas and numpy
import pandas as pd
import numpy as np

# creating simple array
data = np.array(['g', 'e', 'e', 'k', 's', 'f',
                 'o', 'r', 'g', 'e', 'e', 'k', 's'])
ser = pd.Series(data)

# retrieve the first element
```

```
print(ser[-10:])
```

Output:

```
3     k
4     s
5     f
6     o
7     r
8     g
9     e
10    e
11    k
12    s
dtype: object
```

```
# importing pandas module
import pandas as pd

# making data frame
df = pd.read_csv("nba.csv")

ser = pd.Series(df['Name'])
ser.head(10)
```

```
0    Avery Bradley
1      Jae Crowder
2     John Holland
3      R.J. Hunter
4    Jonas Jerebko
5     Amir Johnson
6    Jordan Mickey
7     Kelly Olynyk
8     Terry Rozier
9     Marcus Smart
Name: Name, dtype: object
```

Now we access first 5 elements of series.

```
# get first five names
ser[:5]
```

Output:

```
0    Avery Bradley
1      Jae Crowder
2     John Holland
3      R.J. Hunter
4    Jonas Jerebko
Name: Name, dtype: object
```

Access an Element in Pandas Using Label

In order to access an element from series, we have to set values by index label. A Series is like a fixed-size dictionary in that you can get and set values by index label. Here, we will access an element in Pandas using label.

Accessing a Single Element Using index Label

In this example, a Pandas Series 'ser' is created from a NumPy array 'data' with custom indices provided. The element at index 16 is accessed and printed using `print(ser[16])`.

```
# import pandas and numpy
import pandas as pd
import numpy as np

# creating simple array
data = np.array(['g', 'e', 'e', 'k', 's', 'f',
```

```
            'o', 'r', 'g', 'e', 'e', 'k', 's'])
ser = pd.Series(data, index=[10, 11, 12, 13, 14,
                             15, 16, 17, 18, 19, 20, 21, 22])

# accessing a element using index element
print(ser[16])
```

Output:

```
o
```

Accessing a Multiple Element Using index Label

In this example, a Pandas Series 'ser' is created from a NumPy array 'data' with custom indices provided. Multiple elements at indices 10, 11, 12, 13, and 14 are accessed and printed using `print(ser[[10, 11, 12, 13, 14]])`.

```
# import pandas and numpy
import pandas as pd
import numpy as np

# creating simple array
data = np.array(['g', 'e', 'e', 'k', 's', 'f',
            'o', 'r', 'g', 'e', 'e', 'k', 's'])
ser = pd.Series(data, index=[10, 11, 12, 13, 14,
                             15, 16, 17, 18, 19, 20, 21, 22])

# accessing a multiple element using
# index element
print(ser[[10, 11, 12, 13, 14]])
```

Output:

```
10    g
11    e
12    e
13    k
14    s
dtype: object
```

Access Multiple Elements by Providing Label of Index

In this example, a Pandas Series 'ser' is created using NumPy's arange() function with values from 3 to 8 and custom indices. Elements at indices 'a', 'd' are accessed and printed using `print(ser[['a', 'd']])`.

```
# importing pandas and numpy
import pandas as pd
import numpy as np

ser = pd.Series(np.arange(3, 9), index=['a', 'b', 'c', 'd', 'e', 'f'])

print(ser[['a', 'd']])
```

Output:

```
a    3.0
d    6.0
dtype: float64
```

Accessing a Multiple Element Using Index Label in nba.csv File

In this example, the Pandas module is imported, and a DataFrame 'df' is created by reading data from a CSV file named "nba.csv" using `pd.read_csv`. A Pandas Series 'ser' is then created by selecting the 'Name' column from the DataFrame. Finally, the first 10 elements of the series are accessed and displayed using `ser.head(10)`.

```
# importing pandas module
import pandas as pd

# making data frame
df = pd.read_csv("nba.csv")
```

```
ser = pd.Series(df['Name'])
ser.head(10)
```

```
0     Avery Bradley
1       Jae Crowder
2      John Holland
3        R.J. Hunter
4     Jonas Jerebko
5      Amir Johnson
6     Jordan Mickey
7      Kelly Olynyk
8      Terry Rozier
9     Marcus Smart
Name: Name, dtype: object
```

Now we access an multiple element using index label.

```
ser[[0, 3, 6, 9]]
```

Output:

```
0     Avery Bradley
3       R.J. Hunter
6     Jordan Mickey
9     Marcus Smart
Name: Name, dtype: object
```

Binary operations on Pandas DataFrame and Series

Binary operations involve applying mathematical or logical operations on two objects, typically DataFrames or Series, to produce a new result. Let's learn how binary operations work in Pandas, focusing on their usage with DataFrames and Series.

The most common binary operations include:

- **Arithmetic operations**: Addition, subtraction, multiplication, division, etc.
- **Comparison operations**: Equal to, not equal to, greater than, less than, etc.
- **Logical operations**: And, or, etc.

Pandas makes it easy to perform these operations element-wise (i.e., on a per-row or per-column basis), which is particularly useful when working with large datasets.

Binary Operations on Pandas Series

1. Arithmetic Operations on Series

Arithmetic operations between two Series is applied element-wise. The index labels must align for the operation to work. If the indexes don't match, Pandas will fill in missing values with NaN.

Example: Adding Two Series

```
import pandas as pd
s1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
s2 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# Adding the two Series
result = s1 + s2
print(result)
```

Output

```
a  11
b  22
c  33
dtype: int64
```

2. Comparison Operations on Series

Comparison operations return a Series of boolean values, indicating whether the comparison is True or False for each corresponding element.

Example: Checking Equality

```
import pandas as pd
s1 = pd.Series([10, 20, 30])
s2 = pd.Series([10, 25, 30])

# Comparing the two Series
result = s1 == s2
print(result)
```

Output

```
0    True
1   False
2    True
dtype: bool
```

Binary Operations on Pandas DataFrame

1. Arithmetic Operations on DataFrames

Similar to Series, DataFrame arithmetic operations apply element-wise between two DataFrames.

Note: The DataFrames must have the same shape or matching indexes and columns.

Example: Subtracting DataFrames

```
import pandas as pd
df1 = pd.DataFrame({'A': [10, 20, 30], 'B': [40, 50, 60]})
df2 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Subtracting the DataFrames
result = df1 - df2
print(result)
```

Output

```
   A   B
0  9  36
1  18  45
2  27  54
```

2. Comparison Operations on DataFrames

Like Series, comparison operations on DataFrames return a DataFrame of boolean values. These boolean values indicate whether the corresponding elements are equal or satisfy other comparison conditions.

Example: Checking Greater Than

```
import pandas as pd
df1 = pd.DataFrame({'A': [10, 20, 30], 'B': [40, 50, 60]})
df2 = pd.DataFrame({'A': [5, 15, 35], 'B': [30, 60, 55]})

# Checking if elements of df1 are greater than df2
result = df1 > df2
print(result)
```

Output

```
     A     B
0  True   True
```

```
1  True  False
2 False  True
```

Logical Operations on DataFrame and Series

Pandas also supports logical operations (AND, OR, etc.) on DataFrames and Series. These are commonly used for filtering and applying conditions.

Example: Logical AND on Series

```
import pandas as pd
s1 = pd.Series([True, False, True])
s2 = pd.Series([False, False, True])

# Applying logical AND
result = s1 & s2
print(result)
```

Output

```
0   False
1   False
2    True
dtype: bool
```

Handling Missing Data in Binary Operations

When performing binary operations on DataFrames or Series, missing data (NaN) can affect the results. Pandas handles missing data based on the operation:

- Arithmetic operations involving NaN will generally return NaN (e.g., NaN + 1 = NaN).
- Logical operations involving NaN might return False or True, depending on the operation.

Example: Arithmetic with NaN

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})
df2 = pd.DataFrame({'A': [1, None, 3], 'B': [None, 5, 6]})

# Adding the DataFrames
result = df1 + df2
print(result)
```

Output

```
   A    B
0  2.0   NaN
1  NaN   NaN
2  NaN  12.0
```

Pandas Series Index Attribute

Pandas Series is a **one-dimensional labeled array** capable of holding any data type (integers, strings, floats, etc.), with each element having an associated label known as its **index**. The **Series.index attribute** in Pandas allows users to get or set the index labels of a Series object, enhancing data accessibility and retrieval efficiency. **Example:**

```
import pandas as pd

data = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
```

```
# Accessing the index
print("Original Index:", data.index)

# Modifying the index
data.index = ['w', 'x', 'y', 'z']
print("Modified Series:\n", data)
```

Output

```
Original Index: Index(['a', 'b', 'c', 'd'], dtype='object')

Modified Series:

 w   10

x   20

y   30

z   40

dtype: int64
```

Explanation: This code creates a Pandas Series with custom index labels ('a', 'b', 'c', 'd') and retrieves the index using **data.index.** It then updates the index to ('w', 'x', 'y', 'z').

Syntax

Series.index # Access index labels
Series.index = new_index # Modify index labels

Parameter: This method does not take any parameter.

Returns: Index labels of the Series.

Functionality:

- Retrieves the current index labels of the Series.
- Can be used to set new index labels.
- Supports both unique and duplicate index labels.
- Useful for locating elements efficiently within a Series.

Examples of Pandas Series Index() Attribute

Example 1. Assigning Duplicate Index Labels

Pandas allows assigning duplicate index labels, which can be useful in cases where multiple elements share the same category.

```
import pandas as pd

series = pd.Series(['New York', 'Chicago', 'Toronto', 'Lisbon'])

# Creating the row axis labels
series.index = ['City 1', 'City 1', 'City 3', 'City 3']
print(series)
```

Output

```
City 1    New York

City 1     Chicago

City 3    Toronto

City 3     Lisbon

dtype: object
```

Explanation: Even with duplicate labels (**'City 1' and 'City 3'** appearing twice), Pandas maintains the Series structure and ensures data integrity.

Example 2. Retrieving Index Labels

The **Series.index attribute** can also be used to retrieve the current index labels of a Series.

```
import pandas as pd
```

```
Date = ['1/1/2018', '2/1/2018', '3/1/2018', '4/1/2018']
idx_name = ['Day 1', 'Day 2', 'Day 3', 'Day 4']

sr = pd.Series(data = Date,index = idx_name)
print(sr.index)
```

Output

```
Index(['Day 1', 'Day 2', 'Day 3', 'Day 4'], dtype='object')
```

Explanation: The index labels ('Day 1' to 'Day 4') are assigned to a Series and retrieved using series.index.

Example 3. Resetting Index to Default

If needed, we can reset the index to default integer values.

```
import pandas as pd

Date = ['1/1/2018', '2/1/2018', '3/1/2018', '4/1/2018']
idx_name = ['Day 1', 'Day 2', 'Day 3', 'Day 4']

sr = pd.Series(data = Date,       # Series Data
          index = idx_name   # Index
          )

# Resetting index to default
sr.reset_index(drop=True, inplace=True)
print(sr)
```

Output

```
0   1/1/2018

1   2/1/2018

2   3/1/2018

3   4/1/2018

dtype: object
```

Create a Pandas Series from Array

A **Pandas Series** is a one-dimensional labeled array that stores various data types, including numbers (integers or floats), strings, and Python objects. It is a fundamental data structure in the Pandas library used for efficient data manipulation and analysis. In this guide we will explore two simple methods to create a Pandas **Series** from a NumPy array.

Creating a Pandas Series Without an Index

By default when you create a Series from a NumPy array Pandas automatically assigns a numeric index starting from **0**. Here pd.Series(data) converts the array into a **Pandas Series** automatically assigning an index.

```
import pandas as pd
import numpy as np

data = np.array(['a', 'b', 'c', 'd', 'e'])

s = pd.Series(data)
print(s)
```

Output:

```
0    a
1    b
2    c
3    d
4    e
dtype: object
```

Explanation:

- The default index starts from **0** and increments by **1**.
- The data type (dtype: object) means it stores text values

Creating a Pandas Series With a Custom Index

In this method we specify custom indexes instead of using Pandas' default numerical indexing. This is useful when working with structured data, such as **employee IDs, timestamps, or product codes** where meaningful indexes enhance data retrieval and analysis.

```
import pandas as pd
import numpy as np

data = np.array(['a', 'b', 'c', 'd', 'e'])

s = pd.Series(data, index=[1000, 1001, 1002, 1003, 1004])

print(s)
```

Output:

```
1000    a
1001    b
1002    c
1003    d
1004    e
dtype: object
```

Data Input and Output (I/O)

Pandas offers a variety of functions to read data from and write data to different file formats as given below:

- Read CSV Files with Pandas
- Writing data to CSV Files
- Export Pandas dataframe to a CSV file
- Read JSON Files with Pandas
- Parsing JSON Dataset
- Exporting Pandas DataFrame to JSON File
- Working with Excel Files in Pandas
- Read Text Files with Pandas
- Text File to CSV using Python Pandas

Pandas Read CSV in Python

CSV files are the Comma Separated Files. It allows users to load tabular data into a **DataFrame**, which is a powerful structure for data manipulation and analysis. To access data from the CSV file, we require a function read_csv() from Pandas that retrieves data in the form of the data frame. Here's a quick example to get you started.

Suppose you have a file named people.csv. First, we must import the Pandas library. then using Pandas load this data into a DataFrame as follows:

```
import pandas as pd

# reading csv file
```

```
df = pd.read_csv("people.csv")
df
```

Output:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
3	Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
4	Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon

read_csv() function - Syntax & Parameters

read_csv() function in Pandas is used to read data from CSV files into a Pandas **DataFrame**. A DataFrame is a powerful data structure that allows you to manipulate and analyze tabular data efficiently. CSV files are plain-text files where each row represents a record, and columns are separated by commas (or other delimiters).

Here is the **Pandas read CSV** syntax with its parameters.

Syntax: ***pd.read_csv****(filepath_or_buffer, sep=' ,' , header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)*

Parameters:

- ***filepath_or_buffer****: Location of the csv file. It accepts any string path or URL of the file.*
- ***sep****: It stands for separator, default is ', '.*
- ***header****: It accepts int, a list of int, row numbers to use as the column names, and the start of the data. If no names are passed, i.e., header=None, then, it will display the first column as 0, the second as 1, and so on.*
- ***usecols****: Retrieves only selected columns from the CSV file.*
- ***nrows****: Number of rows to be displayed from the dataset.*
- ***index_col****: If None, there are no index numbers displayed along with records.*
- ***skiprows****: Skips passed rows in the new data frame.*

Features in Pandas read_csv

1. Read specific columns using read_csv

The **usecols parameter** allows to load only specific columns from a CSV file. This reduces memory usage and processing time by importing only the required data.

```
df = pd.read_csv("people.csv", usecols=["First Name", "Email"])
print(df)
```

Output:

```
  First Name                Email
0     Shelby      elijah57@example.net
1    Phillip     bethany14@example.com
2   Kristine      bthompson@example.com
3    Yesenia   kaitlinkaiser@example.com
4       Lori  buchananmanuel@example.net
```

2. Setting an Index Column (index_col)

The index_col parameter **sets one or more columns as the DataFrame index**, making the specified column(s) act as row labels for easier data referencing.

```
df = pd.read_csv("people.csv", index_col="First Name")
print(df)
```

Output:

First Name	Last Name	Sex	Email	Date of birth	Job Title
Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon

Using pd.read_csv, setting columns as the DataFrame index

3. Handling Missing Values Using read_csv

The na_values parameter replaces specified strings (e.g., "N/A", "Unknown") with NaN, enabling consistent handling of missing or incomplete data during analysis.\

```
df = pd.read_csv("people.csv", na_values=["N/A", "Unknown"])
```

We won't got nan values as there is no missing value in our dataset.

4. Reading CSV Files with Different Delimiters

In this example, we will take a CSV file and then add some special characters to see how the **sep** parameter works.

```
import pandas as pd

# Sample data stored in a multi-line string
data = """totalbill_tip, sex:smoker, day_time, size
16.99, 1.01:Female|No, Sun, Dinner, 2
10.34, 1.66, Male, No|Sun:Dinner, 3
21.01:3.5_Male, No:Sun, Dinner, 3
23.68, 3.31, Male|No, Sun_Dinner, 2
24.59:3.61, Female_No, Sun, Dinner, 4
25.29, 4.71|Male, No:Sun, Dinner, 4"""

# Save the data to a CSV file
with open("sample.csv", "w") as file:
  file.write(data)
print(data)
```

Output:

```
totalbill_tip, sex:smoker, day_time, size
16.99, 1.01:Female|No, Sun, Dinner, 2
10.34, 1.66, Male, No|Sun:Dinner, 3
21.01:3.5_Male, No:Sun, Dinner, 3
23.68, 3.31, Male|No, Sun_Dinner, 2
24.59:3.61, Female_No, Sun, Dinner, 4
25.29, 4.71|Male, No:Sun, Dinner, 4
```

The sample data is stored in a multi-line string for demonstration purposes.

- **Separator (sep)**: The sep='[:, |_]' argument allows Pandas to handle multiple delimiters (:, |, _, ,) using a regular expression.
- **Engine**: The engine='python' argument is used because the default C engine does not support regular expressions for delimiters.

```
# Load the CSV file using pandas with multiple delimiters
df = pd.read_csv('sample.csv',
        sep='[:, |_]',  # Define the delimiters
        engine='python')  # Use Python engine for regex separators
```

df

Output:

	totalbill	tip	Unnamed: 2	sex	smoker	Unnamed: 5	day	time	Unnamed: 8	size
16.99	NaN	1.01	Female	No	NaN	Sun	NaN	Dinner	NaN	2.0
10.34	NaN	1.66	NaN	Male	NaN	No	Sun	Dinner	NaN	3.0
21.01	3.50	Male	NaN	No	Sun	NaN	Dinner	NaN	3.0	NaN
23.68	NaN	3.31	NaN	Male	No	NaN	Sun	Dinner	NaN	2.0
24.59	3.61	NaN	Female	No	NaN	Sun	NaN	Dinner	NaN	4.0
25.29	NaN	4.71	Male	NaN	No	Sun	NaN	Dinner	NaN	4.0

5. Using nrows in read_csv()

The nrows parameter limits the number of rows read from a file, enabling quick previews or partial data loading for large datasets. Here, we just display only 5 rows using **nrows parameter**.

```
df = pd.read_csv('people.csv', nrows=3)
df
```

Output:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath

6. Using skiprows in read_csv()

The skiprows parameter skips unnecessary rows at the start of a file, which is useful for ignoring metadata or extra headers that are not part of the dataset.

```
df= pd.read_csv("people.csv")
print("Previous Dataset: ")
print(df)
# using skiprows
df = pd.read_csv("people.csv", skiprows = [4,5])
print("Dataset After skipping rows: ")
print(df)
```

Output:

Previous Dataset:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
3	Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
4	Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon
5	Erin	Day	Male	tconner@example.org	2015-10-28	Management officer
6	Katherine	Buck	Female	conniecowan@example.com	1989-01-22	Analyst
7	Ricardo	Hinton	Male	wyattbishop@example.com	1924-03-26	Hydrogeologist

Dataset After skipping rows:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath

7. Parsing Dates (parse_dates)

The parse_dates parameter converts date columns into datetime objects, simplifying operations like filtering, sorting, or time-based analysis.

```
df = pd.read_csv("people.csv", parse_dates=["Date of birth"])
print(df.info())
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   First Name    5 non-null     object
 1   Last Name     5 non-null     object
 2   Sex           5 non-null     object
 3   Email         5 non-null     object
 4   Date of birth  5 non-null    datetime64[ns]
 5   Job Title     5 non-null     object
dtypes: datetime64[ns](1), object(5)
memory usage: 368.0+ bytes
```

Loading a CSV Data from a URL

Pandas allows you to directly read a CSV file hosted on the internet using the file's URL. This can be incredibly useful when working with datasets shared on websites, cloud storage, or public repositories like GitHub.

```
url = "https://media.TechforTech.org/wp-content/uploads/20241121154629307916/people_data.csv"
df = pd.read_csv(url)
df
```

Output:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
3	Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
4	Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon

Saving a Pandas Dataframe as a CSV

In this article, we will learn how we can export a Pandas DataFrame to a CSV file by using the Pandas to_csv() method. By default, the to csv() method exports DataFrame to a CSV file with row index as the first column and comma as the delimiter.

Table of Content

Here, we are taking sample data to convert DataFrame to CSV.

```
# importing pandas as pd
import pandas as pd

# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]
```

```
# dictionary of lists
dict = {'name': nme, 'degree': deg, 'score': scr}

df = pd.DataFrame(dict)
```

Export CSV to a Working Directory
Here, we simply export a Dataframe to a CSV file using df.to_csv().

```
# saving the dataframe
df.to_csv('file1.csv')
```

Output:

	name	degree	score
0	aparna	MBA	90
1	pankaj	BCA	40
2	sudhir	M.Tech	80
3	Geeku	MBA	98

Saving CSV Without Headers and Index
Here, we are saving the file with no header and no index number.

```
# saving the dataframe
df.to_csv('file2.csv', header=False, index=False)
```

Output:

Save the CSV file to a Specified Location
We can also, save our file at some specific location.

```
# saving the dataframe
df.to_csv(r'C:\Users\Admin\Desktop\file3.csv')
```

Output:

	name	degree	score
0	aparna	MBA	90
1	pankaj	BCA	40
2	sudhir	M.Tech	80
3	Geeku	MBA	98

Write a DataFrame to CSV file using Tab Separator
We can also save our file with some specific separate as we want. i.e, "\t" .

```
import pandas as pd
import numpy as np

users = {'Name': ['Amit', 'Cody', 'Drew'],
   'Age': [20,21,25]}

#create DataFrame
df = pd.DataFrame(users, columns=['Name','Age'])

print("Original DataFrame:")
print(df)
print('Data from Users.csv:')

df.to_csv('Users.csv', sep='\t', index=False,header=True)
new_df = pd.read_csv('Users.csv')

print(new_df)
```

Output:

```
Original DataFrame:
  Name  Age
0 Amit   20
1 Cody   21
2 Drew   25
Data from Users.csv:
  Name\tAge
0 Amit\t20
1 Cody\t21
2 Drew\t25
```

Export Pandas dataframe to a CSV file

When working on a Data Science project one of the key tasks is **data management** which includes data collection, cleaning and storage. Once our data is cleaned and processed it's essential to save it in a structured format for further analysis or sharing.

A **CSV (Comma-Separated Values) file** is a widely used format for storing tabular data. In Python **Pandas** provides an easy-to-use function **to_csv()**to export a DataFrame into a CSV file. This article will walk we through the process with step-by-step examples and customizations.

Creating a Sample DataFrame

Before exporting let's first create a sample DataFrame using Pandas.

```
import pandas as pd

scores = {'Name': ['a', 'b', 'c', 'd'],
      'Score': [90, 80, 95, 20]}

df = pd.DataFrame(scores)
print(df)
```

Output :

```
print (df)
```

```
   Name  Score
0     a     90
1     b     80
2     c     95
3     d     20
```

Now that we have a sample DataFrame, let's export it to a CSV file.

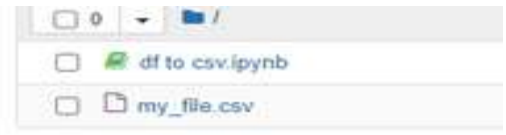
Exporting DataFrame to CSV

1. Basic Export

The simplest way to export a DataFrame to a CSV file is by using the to_csv() function without any additional parameters. This method creates a CSV file where the DataFrame's contents are written as-is.

df.to_csv("your_name.csv")

Output




```
1  ,Name,Score
2  0,a,90
3  1,b,80
4  2,c,95
5  3,d,20
6
```

Customizing the CSV Export

2. Remove Index Column

The **to_csv()** exports the **index** column which represents the row numbers of the DataFrame. If we do not want this extra column in our CSV file we can remove it by setting index=False.

df.to_csv('your_name.csv', index = False)

Output :

```
1  Name,Score
2  a,90
3  b,80
4  c,95
5  d,20
6
```

3. Export only selected columns

In some cases we may not want to export all columns from our DataFrame. The columns parameter in to_csv() allows us to specify which columns should be included in the output file.

df.to_csv("your_name.csv", columns = ['Name'])

Output :

```
1  ,Name
2  0,a
3  1,b
4  2,c
5  3,d
6
```

4. Exclude Header Row

By default the**to_csv()** function includes column names as the first row of the CSV file. However if we need a headerless file e.g., for certain machine learning models or integration with other systems we can set header=False.

df.to_csv('your_name.csv', header = False)

Output :

```
0,a,90
1,b,80
2,c,95
3,d,20

```

5. Handling Missing Values
DataFrames often contain missing values (NaN) which can cause issues in downstream analysis. By default Pandas writes NaN as an empty field but we can customize this behavior using the na_rep parameter.
df.to_csv("your_name.csv", na_rep = 'nothing')

6. Change Column Separator
CSV files use **commas (,)** by default as delimiters to separate values. However in some cases other delimiters may be required such as **tabs ()**, semicolons (;), or pipes (|). Using a different delimiter can make the file more readable or compatible with specific systems.
df.to_csv("your_name.csv", sep ='\t')
Output :

```
	Name	Score
0	a	90
1	b	80
2	c	95
3	d	20

```

How to Read JSON Files with Pandas

JSON (JavaScript Object Notation) store data using key-value pairs. Reading JSON files using Pandas is simple and helpful when you're working with data in .json format. There are mainly three methods to read Json file using Pandas Some of them are:

- Using pd.read_json() Method
- Using JSON Module and pd.json_normalize() Method
- Using pd.Dataframe() Methods

1. Using pd.read_json() to Read JSON Files in Pandas
The pd.read_json() function helps to read JSON data directly into a DataFrame. This method is used when we working with standard JSON structures. If the file is located on a remote server we can also pass the URL instead of a local file path. Let's say you have a file named data.json with the following content:
[
{"id": 1, "name": "Alice", "age": 25},
{"id": 2, "name": "Bob", "age": 30},
{"id": 3, "name": "Charlie", "age": 22}
]
You can read this JSON file using the code below:

```
import pandas as pd

df = pd.read_json('data.json')
print(df.head())
```

Output:

```
   id     name  age
0   1    Alice   25
1   2      Bob   30
2   3  Charlie   22
```

2. Using json Module and pd.json_normalize() method

The json_normalize() is used when we are working with nested JSON structues. JSON from APIs often comes in nested form and this method helps to **flatten** it into a tabular format that's easier to work with in Pandas. This method is helpful when working with real-world JSON responses from APIs.

```
import pandas as pd
import json

data = {"One": {"0": 60, "1": 60, "2": 60, "3": 45, "4": 45, "5": 60},
      "Two": {"0": 110, "1": 117, "2": 103, "3": 109, "4": 117, "5": 102}}

json_data = json.dumps(data)

df_normalize = pd.json_normalize(json.loads(json_data))
print("\nDataFrame using JSON module and `pd.json_normalize()` method:")
df_normalize
```

Output:

```
DataFrame using JSON module and `pd.json_normalize()` method:
```

	One.0	One.1	One.2	One.3	One.4	One.5	Two.0	Two.1	Two.2	Two.3	Two.4	Two.5
0	60	60	60	45	45	60	110	117	103	109	117	102

3. Using pd.DataFrame with a Dictionary

If JSON data is stored as a dictionary we can directly use pd.DataFrame() convert it into a structured DataFrame. This is helpful when you are working with pre-loaded or manually created JSON data in memory.

```
import pandas as pd

df = pd.DataFrame(data)

print(df)
```

Output:

	One	Two
0	60	110
1	60	117
2	60	103
3	45	109
4	45	117
5	60	102

Pandas - Parsing JSON Dataset

JSON (JavaScript Object Notation) is a popular way to store and exchange data especially used in web APIs and configuration files. Pandas provides tools to parse JSON data and convert it into structured DataFrames for analysis. In this guide we will explore various ways to read, manipulate and normalize JSON datasets in Pandas.

Before working with JSON data we need to import pandas. If you're fetching JSON from a web URL or API you'll also need requests.

```
import pandas as pd
import requests
```

Reading JSON Files

To read a JSON file or URL in pandas we use the read_json function. In the below code path_or_buf is the file or web URL to the JSON file.

```
pd.read_json(path_or_buf)
```

Create a DataFrame and Convert It to JSON

if you don't have JSON file then create a small DataFrame and see how to convert it to JSON using different orientations.

- **orient='split':** separates columns, index and data clearly.
- **orient='index':** shows each row as a key-value pair with its index.

```
df = pd.DataFrame([['a', 'b'], ['c', 'd']],
                  index=['row 1', 'row 2'],
                  columns=['col 1', 'col 2'])

print(df.to_json(orient='split'))
print(df.to_json(orient='index'))
```

Output:

```
{"columns":["col 1","col 2"],"index":["row 1","row 2"],"data":[["a","b"],["c","d"]]}
{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}
```

Read the JSON File directly from Web Data

You can fetch JSON data from online sources using the requests library and then convert it to a DataFrame. In the below example it reads and prints JSON data from the specified API endpoint using the pandas library in Python.

- **requests.get(url)** fetches data from the URL.
- **response.json()** converts response to a Python dictionary/list.
- **json_normalize()** converts nested JSON into a flat table.

```
import pandas as pd
import requests

url = 'https://jsonplaceholder.typicode.com/posts'
response = requests.get(url)

data = pd.json_normalize(response.json())
data.head()
```

Output:

	userId	id	title	body
0	1	1	sunt aut facere repellat provident occaecati e...	quia et suscipit\nsuscipit recusandae consequu...
1	1	2	qui est esse	est rerum tempore vitae\nsequi sint nihil repr...
2	1	3	ea molestias quasi exercitationem repellat qui...	et iusto sed quo iure\nvoluptatem occaecati om...
3	1	4	eum et est occaecati	ullam et saepe reiciendis voluptatem adipisci\...
4	1	5	nesciunt quas odio	repudiandae veniam quaerat sunt sed\nalias aut...

Handling Nested JSON in Pandas

Sometimes JSON data has layers like lists or dictionaries inside other dictionaries then it is called as Nested JSON. To turn deeply nested JSON into a table use json_normalize() from pandas making it easier to analyze or manipulate in a table format.

- **json.load(f):** Loads the raw JSON into a Python dictionary.
- **json_normalize(d['programs']):** Extracts the list under the programs key and flattens it into columns.

```
import json
import pandas as pd
from pandas import json_normalize

with open('/content/raw_nyc_phil.json') as f:
  d = json.load(f)

nycphil = json_normalize(d['programs'])
```

```
nycphil.head(3)
```

Output:

	id	orchestra	programID	season	works
0	1001	NY Philharmonic	5001	2023	[{'workTitle': 'Symphony No. 5', 'soloists': [...
1	1002	NY Philharmonic	5002	2023	[{'workTitle': 'Concerto in D', 'soloists': [{...

Exporting Pandas DataFrame to JSON File

Pandas a powerful Python library for data manipulation provides the to_json() function to convert a DataFrame into a JSON file and the read_json() function to read a JSON file into a DataFrame.
In this article we will explore how to export a Pandas DataFrame to a JSON file with detailed explanations and beginner-friendly steps.

Exporting Pandas DataFrame to JSON

To export a Pandas DataFrame to a JSON file we use the **to_json()** function. This function converts the DataFrame into a JSON format making it easy to store and share data. To read the JSON file back into a DataFrame we use the **read_json()** function. Let's explore some examples to understand how they work.

Example 1: Exporting a Simple DataFrame

This example demonstrates how to create a small DataFrame with three rows and three columns and save it as a JSON file.

```
import pandas as pd

df = pd.DataFrame([['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']],
                  index=['row 1', 'row 2', 'row 3'],
                  columns=['col 1', 'col 2', 'col 3'])

df.to_json('file.json', orient='split', compression='infer', index=True)

df = pd.read_json('file.json', orient='split', compression='infer')
print(df)
```

Output :

	col 1	col 2	col3
row 1	a	b	c
row 2	d	e	f
row3	g	h	i

We can see that the DataFrame has been exported as a JSON file.

Example 2: Exporting a More Detailed DataFrame

In this example we create a DataFrame containing employee details such as **ID, Name and Date of Joining**. The JSON file is exported using the split orientation which efficiently organizes the data by storing **indexes, column names and values separately**.

```
import pandas as pd

df = pd.DataFrame(data=[
  ['15135', 'Alex', '25/4/2014'],
  ['23515', 'Bob', '26/8/2018'],
  ['31313', 'Martha', '18/1/2019'],
  ['55665', 'Alen', '5/5/2020'],
```

```
  ['63513', 'Maria', '9/12/2020']],
  columns=['ID', 'NAME', 'DATE OF JOINING'])

df.to_json('file1.json', orient='split', compression='infer')

df = pd.read_json('file1.json', orient='split', compression='infer')
print(df)
```

Output :

	ID	NAME	DATE OF JOINING
0	15135	Alex	25/4/2014
1	23515	Bob	26/8/2018
2	31313	Martha	18/1/2019
3	55665	Alen	5/5/2020
4	63513	Maria	9/12/2020

Working with Excel files using Pandas

Excel sheets are very instinctive and user-friendly, which makes them ideal for manipulating large datasets even for less technical folks. If you are looking for places to learn to manipulate and automate stuff in Excel files using Python, look no further. You are at the right place.
In this article, you will learn how to use Pandas to work with Excel spreadsheets. In this article we will learn about:

- Read Excel File using Pandas in Python
- Installing and Importing Pandas
- Reading multiple Excel sheets using Pandas
- Application of different Pandas functions

Reading Excel File using Pandas in Python
Installating Pandas
To install Pandas in Python, we can use the following command in the command prompt:

```
pip install pandas
```

To install Pandas in Anaconda, we can use the following command in Anaconda Terminal:

```
conda install pandas
```

Importing Pandas
First of all, we need to import the Pandas module which can be done by running the command:

```
import pandas as pd
```

Input File: Let's suppose the Excel file looks like this
Sheet 1:

E17

	A	B	C	D
1	Roll No.	English	Maths	Science
2	1	19	13	17
3	2	14	20	18
4	3	15	18	19
5	4	13	14	14
6	5	17	16	20
7	6	19	13	17
8	7	14	20	18
9	8	15	18	19
10	9	13	14	14
11	10	17	16	20

Sheet 1

Sheet 2:

C13

	A	B	C	D
1	Roll No.	English	Maths	Science
2	1	14	18	20
3	2	11	19	18
4	3	12	18	16
5	4	15	18	19
6	5	13	14	14
7	6	14	18	20
8	7	11	19	18
9	8	12	18	16
10	9	15	18	19
11	10	13	14	14

Sheet 2

Now we can import the Excel file using the read_excel function in Pandas to read Excel file using Pandas in Python. The second statement reads the data from Excel and stores it into a pandas Data Frame which is represented by the variable newData.

```
df = pd.read_excel('Example.xlsx')
print(df)
```

Output:

	Roll No.	English	Maths	Science
0	1	19	13	17
1	2	14	20	18
2	3	15	18	19
3	4	13	14	14
4	5	17	16	20
5	6	19	13	17
6	7	14	20	18
7	8	15	18	19
8	9	13	14	14
9	10	17	16	20

Loading multiple sheets using Concat() method

If there are multiple sheets in the Excel workbook, the command will import data from the first sheet. To make a data frame with all the sheets in the workbook, the easiest method is to create different data frames separately and then concatenate them. The read_excel method takes argument sheet_name and index_col

where we can specify the sheet of which the frame should be made of and index_col specifies the title column, as is shown below:

Example:

The third statement concatenates both sheets. Now to check the whole data frame, we can simply run the following command:

```
file = 'Example.xlsx'
sheet1 = pd.read_excel(file,
                sheet_name = 0,
                index_col = 0)

sheet2 = pd.read_excel(file,
                sheet_name = 1,
                index_col = 0)

# concatinating both the sheets
newData = pd.concat([sheet1, sheet2])
print(newData)
```

Output:

Roll No.	English	Maths	Science
1	19	13	17
2	14	20	18
3	15	18	19
4	13	14	14
5	17	16	20
6	19	13	17
7	14	20	18
8	15	18	19
9	13	14	14
10	17	16	20
1	14	18	20
2	11	19	18
3	12	18	16
4	15	18	19
5	13	14	14
6	14	18	20
7	11	19	18
8	12	18	16
9	15	18	19
10	13	14	14

Head() and Tail() methods in Pandas

To view 5 columns from the top and from the bottom of the data frame, we can run the command. This head() and tail() method also take arguments as numbers for the number of columns to show.

```
print(newData.head())
print(newData.tail())
```

Output:

	English	Maths	Science
Roll No.			
1	19	13	17
2	14	20	18
3	15	18	19
4	13	14	14
5	17	16	20
	English	Maths	Science
Roll No.			
6	14	18	20
7	11	19	18

8	12	18	16
9	15	18	19
10	13	14	14

Shape() method

The shape() method can be used to view the number of rows and columns in the data frame as follows:

```
newData.shape
```

Output:

```
(20, 3)
```

Sort_values() method in Pandas

If any column contains numerical data, we can sort that column using the sort_values() method in pandas as follows:

```
sorted_column = newData.sort_values(['English'], ascending = False)
```

Now, let's suppose we want the top 5 values of the sorted column, we can use the head() method here:

```
sorted_column.head(5)
```

Output:

English	Maths		Science
Roll			No.
1	19	13	17
6	19	13	17
5	17	16	20
10	17	16	20
3	15	18	19

We can do that with any numerical column of the data frame as shown below:

```
newData['Maths'].head()
```

Output:

```
Roll
1    13
2    20
3    18
4    14
5    16
Name: Maths, dtype: int64
```

Pandas Describe() method

Now, suppose our data is mostly numerical. We can get the statistical information like mean, max, min, etc. about the data frame using the describe() method as shown below:

```
newData.describe()
```

Output:

English		Maths	Science
count	20.00000	20.000000	20.000000
mean	14.30000	16.800000	17.500000
std	2.29645	2.330575	2.164304
min	11.00000	13.000000	14.000000
25%	13.00000	14.000000	16.000000
50%	14.00000	18.000000	18.000000
75%	15.00000	18.000000	19.000000
max	19.00000	20.000000	20.000000

This can also be done separately for all the numerical columns using the following command:

```
newData['English'].mean()
```

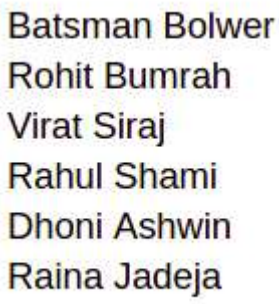
Output:

```
14.3
```

How to Read Text Files with Pandas

In this article, we will discuss how to read text files with pandas in Python. In Python, the Pandas module allows us to load DataFrames from external files and work on them. The dataset can be in different types of files.

Text File Used



Read Text Files with Pandas

Below are the methods by which we can read text files with Pandas:

- Using read_csv()
- Using read_table()
- Using read_fwf()

Read Text Files with Pandas Using read_csv()

We will read the text file with pandas using the read_csv() function. Along with the text file, we also pass separator as a single space (‘ ’) for the space character because, for text files, the space character will separate each field. There are three parameters we can pass to the read_csv() function.

Syntax:

data=pandas.read_csv('filename.txt', sep=' ', header=None, names=["Column1", "Column2"])

Parameters:

- ***filename.txt:*** *As the name suggests it is the name of the text file from which we want to read data.*
- ***sep****: It is a separator field. In the text file, we use the space character(' ') as the separator.*
- ***header:*** *This is an optional field. By default, it will take the first line of the text file as a header. If we use header=None then it will create the header.*
- ***names:*** *We can assign column names while importing the text file by using the names argument.*

```
# Read Text Files with Pandas using read_csv()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame
df = pd.read_csv("gfg.txt", sep=" ")

# display DataFrame
print(df)
```

Output:

```
   Batsman  Bolwer
0   Rohit  Bumrah
1   Virat   Siraj
2   Rahul   Shami
3   Dhoni  Ashwin
4   Raina  Jadeja
```

```
# Read Text Files with Pandas using read_csv()
```

```
# importing pandas
import pandas as pd

# read text file into pandas DataFrame and
# create header
df = pd.read_csv("gfg.txt", sep=" ", header=None)

# display DataFrame
print(df)
```

Output:

```
         0       1
0  Batsman  Bolwer
1    Rohit  Bumrah
2    Virat   Siraj
3    Rahul   Shami
4    Dhoni  Ashwin
5    Raina  Jadeja
```

In the above output, we can see it creates a header starting from number 0. But we can also give names to the header. In this example, we will see how to create a header with a name using pandas.

```
# Read Text Files with Pandas using read_csv()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame and create
# header with names
df = pd.read_csv("gfg.txt", sep=" ", header=None,
          names=["Team1", "Team2"])

# display DataFrame
print(df)
```

Output:

```
     Team1   Team2
0  Batsman  Bolwer
1    Rohit  Bumrah
2    Virat   Siraj
3    Rahul   Shami
4    Dhoni  Ashwin
5    Raina  Jadeja
```

Read Text Files with Pandas Using read_table()

We can read data from a text file using read_table() in pandas. This function reads a general delimited file to a DataFrame object. This function is essentially the same as the read_csv() function but with the delimiter = '\t', instead of a comma by default. We will read data with the read_table function making separator equal to a single space(' ').

Syntax: *data=pandas.read_table('filename.txt', delimiter = ' ')*

Parameters:

- ***filename.txt:*** *As the name suggests it is the name of the text file from which we want to read data.*

Example: In this example, we are using read_table() function to read the table.

```
# Read Text Files with Pandas using read_table()

# importing pandas
import pandas as pd
```

```
# read text file into pandas DataFrame
df = pd.read_table("gfg.txt", delimiter=" ")

# display DataFrame
print(df)
```

Output:

```
  Batsman  Bolwer
0   Rohit  Bumrah
1   Virat   Siraj
2   Rahul   Shami
3   Dhoni  Ashwin
4   Raina  Jadeja
```

Read Text Files with Pandas Using read_fwf()

The fwf in the read_fwf() function stands for fixed-width lines. We can use this function to load DataFrames from files. This function also supports text files. We will read data from the text files using the read_fwf() function with pandas. It also supports optionally iterating or breaking the file into chunks. Since the columns in the text file were separated with a fixed width, this read_fwf() read the contents effectively into separate columns.

Syntax: *data=pandas.read_fwf('filename.txt')*

Parameters:

- ***filename.txt:*** *As the name suggests it is the name of the text file from which we want to read data*.

Example: In this example, we are using read_fwf to read the data.

```
# Read Text Files with Pandas using read_fwf()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame
df = pd.read_fwf("gfg.txt")

# display DataFrame
print(df)
```

Output:

```
  Batsman Bolwer
0   Rohit Bumrah
1   Virat Siraj
2   Rahul Shami
3   Dhoni Ashwin
4   Raina Jadeja
```

Convert Text File to CSV using Python Pandas

Converting Text File to CSV using Python Pandas refers to the process of transforming a plain text file (often with data separated by spaces, tabs, or other delimiters) into a structured CSV (Comma Separated Values) file using the Python Pandas library.

In this article we will walk you through multiple methods to convert a text file into a CSV file using Python's **Pandas** library.

Convert a Simple Text File to CSV

Consider a text file named TechforTech.txt containing structured data. Python will read this data and create a DataFrame where each row corresponds to a line in the text file and each column represents a field in a single line.

Original Text File

```
import pandas as pd

df1 = pd.read_csv("TechforTech.txt")

df1.to_csv('TechforTech.csv',
           index = None)
```

Output:

	A	B	C	D	E	F
1	S.NO	NAME	ADDRESS	EMAIL	PHONE No.	
2	1	Rohan	Lucknow	roh@gmail.com	1234567890	
3	2	Ritik	Delhi	ritik@yahoo.com	2345678901	
4	3	Prerit	Jaipur	pat@gmail.com	345678901	
5						

Handling Text Files Without Headers

If the column heading are not given and the text file looks like. In this case you need to manually define the column headers while converting to CSV:

Text File without headers

```
import pandas as pd

websites = pd.read_csv("TechforTech.txt"
                ,header = None)

websites.columns = ['Name', 'Type', 'Website']

websites.to_csv('TechforTech.csv',
           index = None)
```

Handling Custom Delimiters

Some text files use custom delimiters instead of commas. Suppose the file uses / as a separator. To handle this specify the **delimiter** while reading the file

```
import pandas as pd

account = pd.read_csv("TechforTech.txt",
                delimiter = '/')

account.to_csv('TechforTech.csv',
           index = None)
```

Output:

	A	B	C	D	E	F	G	H	I
1	account_number	name	item_code	category	quantity	unit price	net_price	date	
2	296809	Carroll PLC	QN-82852	Belt	13	44.48	578.24	9/27/2014 7:13	
3	98022	Heidenreich-Bosco	MJ-21460	Shoes	19	53.62	1018.78	7/29/2014 2:10	
4	563905	Kerluke, Reilly and Bechtelar	AS-93055	Shirt	12	24.16	289.92	3/1/2014 10:51	
5	93356	Waters-Walker	AS-93055	Shirt	5	82.68	413.4	11/17/2013 20:41	
6									
7									

Data Cleaning in Pandas

Data cleaning is an essential step in data preprocessing to ensure accuracy and consistency. Here are some articles to know more about it:

- Handling Missing Data
- Removing Duplicates
- Pandas Change Datatype

- Drop Empty Columns in Pandas
- String manipulations in Pandas
- String methods in Pandas
- Detect Mixed Data Types and Fix it

Working with Missing Data in Pandas

In Pandas, missing data occurs when some values are missing or not collected properly and these missing values are represented as:

- **None**: A Python object used to represent missing values in object-type arrays.
- **NaN**: A special floating-point value from NumPy which is recognized by all systems that use IEEE floating-point standards.

In this article we see how to detect, handle and fill missing values in a DataFrame to keep the data clean and ready for analysis.

Checking Missing Values in Pandas

Pandas provides two important functions which help in detecting whether a value is NaN helpful in making data cleaning and preprocessing easier in a DataFrame or Series are given below :

1. Using isnull()

isnull() returns a DataFrame of Boolean value where True represents missing data (NaN). This is simple if we want to find and fill missing data in a dataset.

Example 1: Finding Missing Values in a DataFrame

We will be using Numpy and Pandas libraries for this implementation**.**

```
import pandas as pd
import numpy as np

d = {'First Score': [100, 90, np.nan, 95],
     'Second Score': [30, 45, 56, np.nan],
     'Third Score': [np.nan, 40, 80, 98]}
df = pd.DataFrame(d)

mv = df.isnull()

print(mv)
```

	First Score	Second Score	Third Score
0	False	False	True
1	False	False	False
2	True	False	False
3	False	True	False

Output

Example 2: Filtering Data Based on Missing Values

Here we used random Employee dataset. The isnull() function is used over the "Gender" column in order to filter and print out rows containing missing gender data.

You can download the csv file from here

```
import pandas as pd
d = pd.read_csv("/content/employees.csv")

bool_series = pd.isnull(d["Gender"])
missing_gender_data = d[bool_series]
print(missing_gender_data)
```

Output

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
27	Scott	NaN	7/11/1991	6:58 PM	122367	5.218	False	Legal
31	Joyce	NaN	2/20/2005	2:40 PM	88657	12.752	False	Product
41	Christine	NaN	6/28/2015	1:08 AM	66582	11.308	True	Business Development
49	Chris	NaN	1/24/1980	12:13 PM	113590	3.055	False	Sales
51	NaN	NaN	12/17/2011	8:29 AM	41126	14.009	NaN	Sales
53	Alan	NaN	3/3/2014	1:28 PM	40341	17.578	True	Finance
60	Paula	NaN	11/23/2005	2:01 PM	48866	4.271	False	Distribution
64	Kathleen	NaN	4/11/1990	6:46 PM	77834	18.771	False	Business Development
69	Irene	NaN	7/14/2015	4:31 PM	100863	4.382	True	Finance
70	Todd	NaN	6/10/2003	2:26 PM	84692	6.617	False	Client Services
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
939	Ralph	NaN	7/28/1995	6:53 PM	70635	2.147	False	Client Services
945	Gerald	NaN	4/15/1989	12:44 PM	93712	17.426	True	Distribution
961	Antonio	NaN	6/18/1989	9:37 PM	103050	3.050	False	Legal
972	Victor	NaN	7/28/2006	2:49 PM	76381	11.159	True	Sales
985	Stephen	NaN	7/10/1983	8:10 PM	85668	1.909	False	Legal
989	Justin	NaN	2/10/1991	4:58 PM	38344	3.794	False	Legal
995	Henry	NaN	11/23/2014	6:09 AM	132483	16.655	False	Distribution

145 rows × 8 columns

2. Using isna()

isna() returns a DataFrame of Boolean values where True indicates missing data (NaN). It is used to detect missing values just like isnull().

Example: Finding Missing Values in a DataFrame

```
import pandas as pd
import numpy as np

data = {'Name': ['Amit', 'Sita', np.nan, 'Raj'],
    'Age': [25, np.nan, 22, 28]}

df = pd.DataFrame(data)

# Check for missing values using isna()
print(df.isna())
```

Output:

```
    Name    Age
0  False  False
1  False   True
2   True  False
3  False  False
```

Using isna()

3. Checking for Non-Missing Values Using notnull()

notnull() function returns a DataFrame with Boolean values where True indicates non-missing (valid) data. This function is useful when we want to focus only on the rows that have valid, non-missing values.

Example 1: Identifying Non-Missing Values in a DataFrame

```
import pandas as pd
import numpy as np
```

```
d = {'First Score': [100, 90, np.nan, 95],
     'Second Score': [30, 45, 56, np.nan],
     'Third Score': [np.nan, 40, 80, 98]}
df = pd.DataFrame(d)

nmv = df.notnull()

print(nmv)
```

Output

	First Score	Second Score	Third Score
0	True	True	False
1	True	True	True
2	False	True	True
3	True	False	True

Example 2: Filtering Data with Non-Missing Values

notnull() function is used over the "Gender" column in order to filter and print out rows containing missing gender data.

```
import pandas as pd
d = pd.read_csv("/content/employees.csv")

nmg = pd.notnull(d["Gender"])

nmgd= d[nmg]

display(nmgd)
```

Output

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	8/6/1993	12:42 PM	97308	6.945	True	Marketing
1	Thomas	Male	3/31/1996	6:53 AM	61933	4.170	True	NaN
2	Maria	Female	4/23/1993	11:17 AM	130590	11.858	False	Finance
3	Jerry	Male	3/4/2005	1:00 PM	138705	9.340	True	Finance
4	Larry	Male	1/24/1998	4:47 PM	101004	1.389	True	Client Services
5	Dennis	Male	4/18/1987	1:35 AM	115163	10.125	False	Legal
6	Ruby	Female	8/17/1987	4:20 PM	65476	10.012	True	Product
7	NaN	Female	7/20/2015	10:43 AM	45906	11.598	NaN	Finance
8	Angela	Female	11/22/2005	6:29 AM	95570	18.523	True	Engineering
9	Frances	Female	8/8/2002	6:51 AM	139852	7.524	True	Business Development
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
994	George	Male	6/21/2013	5:47 PM	98874	4.479	True	Marketing
996	Phillip	Male	1/31/1984	6:30 AM	42392	19.675	False	Finance
997	Russell	Male	5/20/2013	12:39 PM	96914	1.421	False	Product
998	Larry	Male	4/20/2013	4:45 PM	60500	11.985	False	Business Development
999	Albert	Male	5/15/2012	6:24 PM	129949	10.169	True	Sales

855 rows × 8 columns

Filling Missing Values in Pandas

Following functions allow us to replace missing values with a specified value or use interpolation methods to find the missing data.

1. Using fillna()

fillna() used to replace missing values (NaN) with a given value. Lets see various example for this.

Example 1: Fill Missing Values with Zero

```
import pandas as pd
import numpy as np

d = {'First Score': [100, 90, np.nan, 95],
     'Second Score': [30, 45, 56, np.nan],
     'Third Score': [np.nan, 40, 80, 98]}
df = pd.DataFrame(d)

df.fillna(0)
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	0.0
1	90.0	45.0	40.0
2	0.0	56.0	80.0
3	95.0	0.0	98.0

Fill with Previous Value (Forward Fill)

The pad method is used to fill missing values with the previous value.

```
df.fillna(method='pad')
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	90.0	56.0	80.0
3	95.0	56.0	98.0

Fill with Next Value (Backward Fill)

The bfill function is used to fill it with the next value.

```
df.fillna(method='bfill')
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	40.0
1	90.0	45.0	40.0
2	95.0	56.0	80.0
3	95.0	NaN	98.0

Example 4: Fill NaN Values with 'No Gender'

```
import pandas as pd
import numpy as np
d = pd.read_csv("/content/employees.csv")

d[10:25]
```

Output

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN
24	John	Male	7/1/1992	10:08 PM	97950	13.873	False	Client Services

Now we are going to fill all the null values in Gender column with "No Gender"

```
d["Gender"].fillna('No Gender', inplace = True)
d[10:25]
```

Output

10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	No Gender	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	No Gender	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN
24	John	Male	7/1/1992	10:08 PM	97950	13.873	False	Client Services

2. Using replace()

Use **replace()** function to replace NaN values with a specific value.

Example

```
import pandas as pd
import numpy as np

data = pd.read_csv("/content/employees.csv")
data[10:25]
```

Output

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN

Now, we are going to replace the all NaN value in the data frame with -99 value.

```
data.replace(to_replace=np.nan, value=-99)
```

Output

10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	-99
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	-99	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	-99	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	-99	Male	6/14/2012	4:19 PM	125792	5.042	-99	-99
24	John	Male	7/1/1992	10:08 PM	97950	13.873	False	Client Services

3. Using interpolate()

The **interpolate()** function fills missing values using interpolation techniques such as the linear method.

Example

```
import pandas as pd

df = pd.DataFrame({"A": [12, 4, 5, None, 1],
                   "B": [None, 2, 54, 3, None],
                   "C": [20, 16, None, 3, 8],
                   "D": [14, 3, None, None, 6]})
print(df)
```

Output

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	NaN	NaN
3	NaN	3.0	3.0	NaN
4	1.0	NaN	8.0	6.0

```
df.interpolate(method ='linear', limit_direction ='forward')
```

Output

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	9.5	4.0
3	3.0	3.0	3.0	5.0
4	1.0	3.0	8.0	6.0

Dropping Missing Values in Pandas

The **dropna()** function used to removes rows or columns with NaN values. It can be used to drop data based on different conditions.

1. Dropping Rows with At Least One Null Value

Remove rows that contain at least one missing value.

Example

```
import pandas as pd
import numpy as np

dict = {'First Score': [100, 90, np.nan, 95],
     'Second Score': [30, np.nan, 45, 56],
     'Third Score': [52, 40, 80, 98],
     'Fourth Score': [np.nan, np.nan, np.nan, 65]}
df = pd.DataFrame(dict)

df.dropna()
```

	First Score	Second Score	Third Score	Fourth Score
3	95.0	56.0	98	65.0

2. Dropping Rows with All Null Values

We can drop rows where all values are missing using dropna(how='all').

Example

```
dict = {'First Score': [100, np.nan, np.nan, 95],
     'Second Score': [30, np.nan, 45, 56],
     'Third Score': [52, np.nan, 80, 98],
     'Fourth Score': [np.nan, np.nan, np.nan, 65]}
df = pd.DataFrame(dict)

df.dropna(how='all')
```

Output

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

3. Dropping Columns with At Least One Null Value

To remove columns that contain at least one missing value we use dropna(axis=1).

Example

```
dict = {'First Score': [100, np.nan, np.nan, 95],
     'Second Score': [30, np.nan, 45, 56],
     'Third Score': [52, np.nan, 80, 98],
     'Fourth Score': [60, 67, 68, 65]}
df = pd.DataFrame(dict)

df.dropna(axis=1)
```

Output

	Fourth Score
0	60
1	67
2	68
3	65

4. Dropping Rows with Missing Values in CSV Files

When working with CSV files, we can drop rows with missing values using dropna().

Example

```
import pandas as pd
d = pd.read_csv("/content/employees.csv")

nd = d.dropna(axis=0, how='any')

print("Old data frame length:", len(d))
print("New data frame length:", len(nd))
print("Rows with at least one missing value:", (len(d) - len(nd)))
```

Pandas dataframe.drop_duplicates()

The drop_duplicates() method in Pandas is designed to remove duplicate rows from a DataFrame based on all columns or specific ones. By default, it scans the entire DataFrame and retains the first occurrence of each row and removes any duplicates that follow. In this article, we will see how to use the drop_duplicates() method and its examples.

Let's start with a basic example to see how drop_duplicates() works.

```
import pandas as pd

data = {
  "Name": ["Alice", "Bob", "Alice", "David"],
  "Age": [25, 30, 25, 40],
  "City": ["NY", "LA", "NY", "Chicago"]
}

df = pd.DataFrame(data)
```

```
print("Original DataFrame:")
print(df)

df_cleaned = df.drop_duplicates()

print("\nModified DataFrame (no duplicates)")
print(df_cleaned)
```

Syntax:
DataFrame.drop_duplicates(subset=None, keep='first', inplace=False)
Parameters:
1. subset: Specifies the columns to check for duplicates. If not provided all columns are considered.
2. keep: Finds which duplicate to keep:

- **'first' (default)**: Keeps the first occurrence, removes subsequent duplicates.
- **'last'**: Keeps the last occurrence and removes previous duplicates.
- **False**: Removes all occurrences of duplicates.

3. inplace: If True it modifies the original DataFrame directly. If False (default), returns a new DataFrame.
Return type: Method returns a new DataFrame with duplicates removed unless inplace=True.
Examples
Below are some examples of dataframe.drop_duplicates() method:
1. Dropping Duplicates Based on Specific Columns
We can target duplicates in specific columns using the subset parameter. This is useful when some columns are more relevant for identifying duplicates.

```
import pandas as pd

df = pd.DataFrame({
  'Name': ['Alice', 'Bob', 'Alice', 'David'],
  'Age': [25, 30, 25, 40],
  'City': ['NY', 'LA', 'SF', 'Chicago']
})

df_cleaned = df.drop_duplicates(subset=["Name"])

print(df_cleaned)
```

2. Keeping the Last Occurrence of Duplicates
By default drop_duplicates() retains the first occurrence of duplicates. If we want to keep the last occurrence we can use **keep='last'.**

```
import pandas as pd

df = pd.DataFrame({
  'Name': ['Alice', 'Bob', 'Alice', 'David'],
  'Age': [25, 30, 25, 40],
  'City': ['NY', 'LA', 'NY', 'Chicago']
})

df_cleaned= df.drop_duplicates(keep='last')
print(df_cleaned)
```

3. Dropping All Duplicates
If we want to remove all rows that are duplicates, we can set **keep=False.**

```
import pandas as pd

df = pd.DataFrame({
  'Name': ['Alice', 'Bob', 'Alice', 'David'],
  'Age': [25, 30, 25, 40],
  'City': ['NY', 'LA', 'NY', 'Chicago']
```

```
})
df_cleaned = df.drop_duplicates(keep=False)
print(df_cleaned)
```

4. Modifying the Original DataFrame Directly

If we want to modify the DataFrame in place without creating a new DataFrame set **inplace=True.**

```
import pandas as pd

df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Alice', 'David'],
   'Age': [25, 30, 25, 40],
   'City': ['NY', 'LA', 'NY', 'Chicago']
})
df.drop_duplicates(inplace=True)
print(df)
```

5. Dropping Duplicates Based on Partially Identical Columns

Sometimes we might encounter situations where duplicates are not exact rows but have identical values in certain columns. For example after merging datasets we may want to drop rows that have the same values in a subset of columns.

```
import pandas as pd

data = {
   "Name": ["Alice", "Bob", "Alice", "David", "Bob"],
   "Age": [25, 30, 25, 40, 30],
   "City": ["NY", "LA", "NY", "Chicago", "LA"]
}

df = pd.DataFrame(data)

df_cleaned = df.drop_duplicates(subset=["Name", "City"])

print(df_cleaned)
```

Pandas Change Datatype

Using astype() method

```
import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob', 'Eve', 'Charlie'],
      'Age': [25, 30, 22, 35, 28],
      'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
      'Salary': [50000, 55000, 40000, 70000, 48000]}

df = pd.DataFrame(data)

# Convert 'Age' column to float type
df['Age'] = df['Age'].astype(float)
print(df.dtypes)
```

Output

```
Name      object
Age      float64
Gender    object
Salary    int64
```

```
dtype: object
```

Converting a Column to a DateTime Type

Sometimes, a column that contains date information may be stored as a string. You can convert it to the datetime type using the pd.to_datetime() function.

```
# Example: Create a 'Join Date' column as a string
df['Join Date'] = ['2021-01-01', '2020-05-22', '2022-03-15', '2021-07-30', '2020-11-11']

# Convert 'Join Date' to datetime type
df['Join Date'] = pd.to_datetime(df['Join Date'])
print(df.dtypes)
```

Output

```
Name            object
Age              int64
Gender          object
Salary           int64
Join Date    datetime64[ns]
dtype: object
```

Changing Multiple Columns' Data Types

If you need to change the data types of multiple columns at once, you can pass a dictionary to the astype() method, where keys are column names and values are the desired data types.

```
# Convert 'Age' to float and 'Salary' to string
df = df.astype({'Age': 'float64', 'Salary': 'str'})
print(df.dtypes)
```

Output

```
Name      object
Age      float64
Gender    object
Salary    object
dtype: object
```

Drop Empty Columns in Pandas

Cleaning data is an essential step in data analysis. In this guide we will explore different ways to drop empty, null and zero-value columns in a Pandas DataFrame using Python. By the end you'll know how to efficiently clean your dataset using the dropna() and replace() methods.

Understanding dropna()

The dropna() function is a powerful method in Pandas that allows us to remove rows or columns containing missing values (NaN). Depending on the parameters used it can remove rows or columns where at least one value is missing or only those where all values are missing.

Syntax: *DataFrameName.dropna(axis=0, how='any', inplace=False)*

Parameters:

- **axis:** axis takes int or string value for rows/columns. Input can be 0 or 1 for Integer and ‘index’ or ‘columns’ for String.
- **how:** how takes string value of two kinds only (‘any’ or ‘all’). ‘any’ drops the row/column if ANY value is Null and ‘all’ drops only if ALL values are null.
- **inplace:** It is a boolean which makes the changes in the data frame itself if True.

Create a Sample DataFrame:
This is the sample data frame on which we will use to perform different operations.

```
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                   "Gender": ["", "", ""],
                   "Age": [0, 0, 0]})
df['Department'] = np.nan

print(df)
```

Output:

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

Example 1: Remove All Null Value Columns
This method removes columns where **all** values are NaN. If a column is completely empty (contains only NaN values) it is unnecessary for analysis and can be removed using dropna(how='all', axis=1).

```
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                   "Gender": ["", "", ""],
                   "Age": [0, 0, 0]})

df['Department'] = np.nan

display(df)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

Output:

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

	FirstName	Gender	Age
0	Vipul		0
1	Ashish		0
2	Milan		0

Example 2: Replace Empty Strings with Null and Drop Null Columns
If a column contains empty strings we need to replace them with NaN before dropping the column. Empty strings are not automatically recognized as missing values in Pandas so converting them to NaN ensures they can be handled correctly. After conversion we use dropna(how='all', axis=1) to remove columns that are entirely empty.

```
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                   "Gender": ["", "", ""],
                   "Age": [0, 0, 0]})

df['Department'] = np.nan
display(df)

nan_value = float("NaN")
df.replace("", nan_value, inplace=True)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

Output:

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

	FirstName	Age
0	Vipul	0
1	Ashish	0
2	Milan	0

Example 3: Replace Zeros with Null and Drop Null Columns

If columns contain only zero values, we convert them to NaN before dropping them.

```
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                   "Gender": ["", "", ""],
                   "Age": [0, 0, 0]})

df['Department'] = np.nan
display(df)

nan_value = float("NaN")
df.replace(0, nan_value, inplace=True)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

Output:

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

	FirstName	Gender
0	Vipul	
1	Ashish	
2	Milan	

String manipulations in Pandas DataFrame

String manipulation is the process of changing, parsing, splicing, pasting or analyzing strings. As we know that sometimes data in the string is not suitable for manipulating the analysis or get a description of the data. But Python is known for its ability to manipulate strings. In this article we will understand how Pandas provides us the ways to manipulate to modify and process string data-frame using some builtin functions.

Create a String Dataframe using Pandas

First of all we will know ways to create a string dataframe using Pandas.

```
import pandas as pd
import numpy as np

data = {'Names': ['Gulshan', 'Shashank', 'Bablu', 'Abhishek', 'Anand', np.nan, 'Pratap'],
    'City': ['Delhi', 'Mumbai', 'Kolkata', 'Delhi', 'Chennai', 'Bangalore', 'Hyderabad']}

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

Output:

```
      Names       City
0   Gulshan      Delhi
1  Shashank     Mumbai
2     Bablu    Kolkata
3  Abhishek      Delhi
4     Anand    Chennai
5       NaN  Bangalore
6    Pratap  Hyderabad
```

Change Column Datatype in Pandas

To change the type of the created dataframe to string type. we can do this with the help of **.astype()** . Let's have a look at them in the below example

```
print(df.astype('string'))
```

Output:

```
      Names       City
0   Gulshan      Delhi
1  Shashank     Mumbai
2     Bablu    Kolkata
3  Abhishek      Delhi
4     Anand    Chennai
5       NaN  Bangalore
6    Pratap  Hyderabad
```

String Manipulations in Pandas

Now we see the string manipulations inside a Pandas Dataframe, so first create a Dataframe and manipulate all string operations on this single data frame below so that everyone can get to know about it easily.

Example:

```
import pandas as pd
import numpy as np

data = {'Names': ['Gulshan', 'Shashank', 'Bablu', 'Abhishek', 'Anand', np.nan, 'Pratap'],
    'City': ['Delhi', 'Mumbai', 'Kolkata', 'Delhi', 'Chennai', 'Bangalore', 'Hyderabad']}

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

Output:

```
      Names       City
0   Gulshan      Delhi
1  Shashank     Mumbai
2     Bablu    Kolkata
3  Abhishek      Delhi
4     Anand    Chennai
5       NaN  Bangalore
6    Pratap  Hyderabad
```

Let's have a look at various methods provided by this library for string manipulations.

- **lower():** Converts all uppercase characters in strings in the DataFrame to lower case and returns the lowercase strings in the result.

```
print(df['Names'].str.lower())
```

Output:

```
0     gulshan
1    shashank
2       bablu
3    abhishek
4       anand
5         NaN
6      pratap
Name: Names, dtype: object
```

- **upper():** Converts all lowercase characters in strings in the DataFrame to upper case and returns the uppercase strings in result.

```
print(df['Names'].str.upper())
```

Output:

```
0     GULSHAN
1    SHASHANK
2       BABLU
3    ABHISHEK
4       ANAND
5         NaN
6      PRATAP
Name: Names, dtype: object
```

- **strip():** If there are spaces at the beginning or end of a string, we should trim the strings to eliminate spaces using strip() or remove the extra spaces contained by a string in DataFrame.

```
print(df['Names'].str.strip())
```

Output:

```
0     Gulshan
1    Shashank
2       Bablu
3    Abhishek
4       Anand
5         NaN
6      Pratap
Name: Names, dtype: object
```

- **split(' '):** Splits each string with the given pattern. Strings are split and the new elements after the performed split operation, are stored in a list.

```
df['Split_Names'] = df['Names'].str.split('a')
print(df[['Names', 'Split_Names']])
```

Output:

```
       Names   Split_Names
 0   Gulshan    [Gulsh, n]
 1  Shashank  [Sh, sh, nk]
 2     Bablu      [B, blu]
 3  Abhishek    [Abhishek]
 4     Anand      [An, nd]
 5       NaN           NaN
 6    Pratap    [Pr, t, p]
```

- **len():** With the help of len() we can compute the length of each string in DataFrame & if there is empty data in DataFrame, it returns NaN.

```
print(df['Names'].str.len())
```

Output:

```
0    7.0
1    8.0
2    5.0
3    8.0
4    5.0
5    NaN
6    6.0
Name: Names, dtype: float64
```

- **cat(sep=' '):** It concatenates the data-frame index elements or each string in DataFrame with given separator.

```
print(df)
```

```
print("\nafter using cat:")
print(df['Names'].str.cat(sep=', '))
```

Output:

```
       Names       City   Split_Names
0   Gulshan      Delhi    [Gulsh, n]
1  Shashank     Mumbai  [Sh, sh, nk]
2     Bablu    Kolkata      [B, blu]
3  Abhishek      Delhi    [Abhishek]
4     Anand    Chennai      [An, nd]
5       NaN  Bangalore           NaN
6    Pratap  Hyderabad    [Pr, t, p]

after using cat:
Gulshan, Shashank, Bablu, Abhishek, Anand, Pratap
```

- **get_dummies():** It returns the DataFrame with One-Hot Encoded values like we can see that it returns boolean value 1 if it exists in relative index or 0 if not exists.

```
print(df['City'].str.get_dummies())
```

Output:

```
   Bangalore  Chennai  Delhi  Hyderabad  Kolkata  Mumbai
0          0        0      1          0        0       0
1          0        0      0          0        0       1
2          0        0      0          0        1       0
3          0        0      1          0        0       0
4          0        1      0          0        0       0
5          1        0      0          0        0       0
6          0        0      0          1        0       0
```

- **startswith(pattern):** It returns true if the element or string in the DataFrame Index starts with the pattern.

```
print(df['Names'].str.startswith('G'))
```

Output:

```
0     True
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Names, dtype: object
```

- **endswith(pattern):** It returns true if the element or string in the DataFrame Index ends with the pattern.

```
print(df['Names'].str.endswith('h'))
```

Output:

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Names, dtype: object
```

- **Python replace(a,b):** It replaces the value a with the value b like below in example 'Gulshan' is being replaced by 'Gaurav'.

```
print(df['Names'].str.replace('Gulshan', 'Gaurav'))
```

Output:

```
0      Gaurav
1    Shashank
2       Bablu
3    Abhishek
4       Anand
5         NaN
6      Pratap
Name: Names, dtype: object
```

- **Python repeat(value):** It repeats each element with a given number of times like below in example, there are two appearances of each string in DataFrame.

```
print(df['Names'].str.repeat(2))
```

Output:

```
0      GulshanGulshan
1    ShashankShashank
2          BabluBablu
3    AbhishekAbhishek
4          AnandAnand
5                 NaN
6        PratapPratap
Name: Names, dtype: object
```

- **Python count(pattern):** It returns the count of the appearance of pattern in each element in Data-Frame like below in example it counts 'n' in each string of DataFrame and returns the total counts of 'a' in each string.

```
print(df['Names'].str.count('a'))
```

Output:

```
0    1.0
1    2.0
2    1.0
3    0.0
4    1.0
5    NaN
6    2.0
Name: Names, dtype: float64
```

- **Python find(pattern):** It returns the first position of the first occurrence of the pattern. We can see in the example below that it returns the index value of appearance of character 'a' in each string throughout the DataFrame.

```
print(df['Names'].str.find('a'))
```

Output:

```
0    5.0
1    2.0
2    1.0
3   -1.0
4    2.0
5    NaN
6    2.0
Name: Names, dtype: float64
```

- **findall(pattern):** It returns a list of all occurrences of the pattern. As we can see in below, there is a returned list consisting n as it appears only once in the string.

```
print(df['Names'].str.findall('a'))
```

Output:

```
0       [a]
1    [a, a]
2       [a]
3        []
4       [a]
5       NaN
6    [a, a]
Name: Names, dtype: object
```

- **islower():** It checks whether all characters in each string in the Index of the Data-Frame in lower case or not, and returns a Boolean value.

```
print(df['Names'].str.islower())
```

Output:

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Names, dtype: object
```

- **isupper():** It checks whether all characters in each string in the Index of the Data-Frame in upper case or not, and returns a Boolean value.

```
print(df['Names'].str.isupper())
```

Output:

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Names, dtype: object
```

- **isnumeric():** It checks whether all characters in each string in the Index of the Data-Frame are numeric or not, and returns a Boolean value.

```
print(df['Names'].str.isnumeric())
```

Output:

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Names, dtype: object
```

- **swapcase():** It swaps the case lower to upper and vice-versa. Like in the example below, it converts all uppercase characters in each string into lowercase and vice-versa (lowercase -> uppercase).

```
print(df['Names'].str.swapcase())
```

Output:

```
0     gULSHAN
1    sHASHANK
2       bABLU
3    aBHISHEK
4       aNAND
5         NaN
6      pRATAP
Name: Names, dtype: object
```

String methods in Pandas

String Methods in Pandas

These string methods work in a very efficient way on entire columns of data, so you can modify thousands or even millions of text entries at once without breaking a sweat.

Method	Description
upper()	Converts a string into uppercase
lower()	Converts a string into lowercase
isupper()	Checks whether the character is uppercase or not
islower()	Checks whether the character is lowercase or not
len()	Identifies the length of the string.
startswith()	Returns true if the element starts with the pattern
split()	Splits the string at a particular index or character
find()	Returns the index at where the given string is found
strip()	Strips whitespaces from each string from both sides.
replace()	Replaces a part of the string with another one.

```
import pandas as pd
sports = pd.Series(['Virat', 'azam', 'fiNch', 'ShakiB', 'STOKES', 'KAne'])
print(sports)
```

Output:

```
0     Virat
1      azam
2     fiNch
3    ShakiB
4    STOKES
5      KAne
dtype: object
```

1.) DataFrame.upper()

Convert each string to upper case. This method is useful when normalizing text data for consistency (e.g., converting names or categories to uppercase).

```
print("Upper Case:")
print(sports.str.upper())
```

Output:

```
Upper Case:
0     VIRAT
1      AZAM
2     FINCH
3    SHAKIB
4    STOKES
5      KANE
dtype: object
```

2.) DataFrame.lower()

It converts all characters to lowercase and ensure consistency in text data.

```
print("Lower Case:")
print(s.str.lower())
```

Output:

```
Lower Case:
0     virat
1      azam
2     finch
3    shakib
4    stokes
5      kane
dtype: object
```

3.) DataFrame.isupper()

It returns boolean values based on whether each character present in the string is in upper case or not.

```
print("Checks whether string is in Upper Case:")
print(sports.str.isupper())
```

Output:

```
Checks whether string is in Upper Case:
0    False
1    False
2    False
3    False
4     True
5    False
dtype: bool
```

4.) DataFrame.islower()

It returns boolean values based on whether each character present in the string is in lowercase or not.

```
print("Checks whether string is in Lower Case:")
print(s.str.islower())
```

Output:

```
Checks whether string is in Lower Case:
0    False
1     True
2    False
3    False
4    False
5    False
dtype: bool
```

5.) DataFrame.len()

This function returns the length of each string.

```
print("Length of strings:")
print(sports.str.len())
```

Output:

```
Length of strings:
0    5
1    4
2    5
3    6
4    6
5    4
dtype: int64
```

6.) DataFrame.startswith()

It returns boolean values based on whether the string starts with a certain character sequence or not.

```
print("Checks whether string starts with certain substring:")
print(sports.str.startswith('a'))
```

Output:

```
Checks whether string starts with certain substring:
0    False
1     True
2    False
3    False
4    False
5    False
dtype: bool
```

7.) DataFrame.split()

This function helps to split the string by a certain character or symbols at once.

```
print("Splits string by character 'a':")
print(sports.str.split('a'))
```

Output:

```
Splits string by character 'a':
0     [Vir, t]
1     [, z, m]
2      [fiNch]
3    [Sh, kiB]
4     [STOKES]
5       [KAne]
dtype: object
```

8.) DataFrame.find()

This function finds the index of the occurrence of a certain character sequence.

```
print("Find the index of the searched character or substring:")
print(sports.str.find('a'))
```

Output:

```
Find the index of the searched character or substring:
0    3
1    0
2   -1
3    2
4   -1
5   -1
dtype: int64
```

9.) DataFrame.strip()

It helps to remove the extra trailing spaces from the start and the end.

```
print("Remove extra space from the starting and the end of the string:")
print(s.str.strip())
```

Output:

```
Remove extra space from the starting and the end of the string:
0     Virat
1      azam
2     fiNch
3    ShakiB
4    STOKES
5      KAne
dtype: object
```

10.) DataFrame.replace()

This function helps to remove certain character sequence sometimes which are present in all the strings and is undesired.

```
print("Replace a particular substring by desired pattern:")
print(sports.str.replace('a', ''))
```

Output:

```
Replace a particular substring by desired pattern:
0      Virt
1        zm
2     fiNch
3     ShkiB
4    STOKES
5      KAne
dtype: object
```

Pandas: Detect Mixed Data Types and Fix it

The Python library commonly used for working with data sets and can help users in analyzing, exploring, and manipulating data is known as the Pandas library. When any column of the Pandas data frame doesn't contain a single type of data, either numeric or string, but contains mixed type of data, both numeric as well as string, such column is called a mixed data type column.

Table of Content

What are mixed types in Pandas columns?

As you know, Pandas data frame can have multiple columns, thus when a certain column doesn't have a specified kind of data, i.e., doesn't have a certain data type, but contains mixed data, i.e., numeric as well as string values, then that column is tend to have mixed data type.

For example:

```
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=['Name', 'Age'])
```

Here, the Age column contains string as well as the numeric type of data, the Age column has a mixed data type.

Causes of mixed data types

- Missing Values (NaN)
- Inconsistent Formatting
- Data Entry Errors

Missing Values (NaN):

A floating-point value that represents undefined or unrepresentable data is known as NaN. The most common use case of NaN occurrence is the 0/0 case, which leads to mixed data types and ultimately leads to incorrect results.

Inconsistent Formatting:

The inconsistent formatting in the Pandas data frame is observed due to the cells with wrong format. Thus, it is crucial to transform each cell of column to a correct format.

Data Entry Errors:

There occurs various instances when the user makes a mistake while entering the data in a column in Pandas data frame. It can be any error, entering string data in numeric type column or leaving null value in the column or anything. Such errors can also lead to mixed data types and thus need to be fixed.

How to identify mixed types in Pandas columns

You might have used info() function to detect the data type of Pandas data frame, but using info() function is not possible in case of mixed data types. For detecting the mixed data types, you need to traverse each column of Pandas data frame, and get the data type using api.types.infer_dtypes() function.

Syntax:

```
for column in data_frame.columns:
print(pd.api.types.infer_dtype(data_frame[column]))
```

Here,

- ***data_frame****: It is the Pandas data frame for which you want to detect if it has mixed data types or not.*

Example:

The data frame used in this example to detect mixed data type is as follows:

```
# Python program to detect mixed data types in Pandas data frame

# Import the library Pandas
import pandas as pd

# Create the pandas DataFrame
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=['Name', 'Age'])

# Traverse data frame to detect mixed data types
for column in data_frame.columns:
  print(column,':',pd.api.types.infer_dtype(data_frame[column]))
```

Output:

```
Name : string
Age : mixed-integer
```

How to deal with mixed types in Pandas columns

For fixing the mixed data types in Pandas data frame, you need to convert entire column into one data type. This can be done using **astype**() function or **to_numeric**() function.

Using astype() function:

A crucial function in Pandas which is used to cast an object to a specified data type is known as **astype**() function. In this way, we will see how we can fix mixed data types using **astype**() function.

Syntax:

```
data_frame[column] = data_frame[column].astype(int)
```

Here,

- *data_frame: It is the Pandas data frame for which you want to fix mixed data types.*
- ***column****: It defines all the columns of the Pandas data frame.*
- ***int****: Here, int is the data type in which you want to transform type of each column of Pandas data frame. You can also use str, float, etc. here depending on which data type you want to transform.*

Example:
The data frame used in this example to fix mixed data type is as follows:

```
# Python program to fix mixed data types using astype() in Pandas data frame

# Import the library Pandas
import pandas as pd

# Create the pandas DataFrame
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=['Name', 'Age'])

# Transforming mixed data types to single data type
data_frame[column] = data_frame[column].astype(int)

# Traverse data frame to detect data types after fix
for column in data_frame.columns:
  print(column,':',pd.api.types.infer_dtype(data_frame[column]))
```

Output:

```
Name : string
Age : integer
```

Using to_numeric() function:
The **to_numeric**() function is used to convert an argument to a numeric data type. In this way, we will see how we can fix mixed data types using to_numeric() function.

Syntax:

data_frame[column] = data_frame[column].apply(lambda x: pd.to_numeric(x, errors = 'ignore'))

Here,

- ***data_frame****: It is the Pandas data frame for which you want to fix mixed data types*.
- ***column****: It defines all the columns of the Pandas data frame*.

Example:
The data frame used in this example to fix mixed data type is as follows:

```
# Python program to fix mixed data types using to_numeric() in Pandas data frame

# Import the library Pandas
import pandas as pd

# Create the pandas DataFrame
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=['Name', 'Age'])

# Transforming mixed data types to single data type
data_frame[column] = data_frame[column].apply(lambda x: pd.to_numeric(x, errors = 'ignore'))

# Traverse data frame to detect data types after fix
for column in data_frame.columns:
  print(pd.api.types.infer_dtype(data_frame[column]))
```

Output:

```
Name : string
Age : floating
```

Pandas Operations
We will cover data processing, normalization, manipulation and analysis, along with techniques for grouping and aggregating data. These concepts will help you efficiently clean, transform and analyze datasets. By the end of this section, you'll learn Pandas operations to handle real-world data effectively.

- Data Processing with Pandas.
- Data Normalization in Pandas
- Data Manipulation in Pandas
- Data Analysis using Pandas
- Grouping and Aggregating with Pandas

- Different Types of Joins in Pandas

Data Processing with Pandas

Data Processing is an important part of any task that includes data-driven work. It helps us to provide meaningful insights from the data. As we know Python is a widely used programming language, and there are various libraries and tools available for data processing.
In this article, we are going to see ***Data Processing in Python, Loading, Printing rows and Columns, Data frame summary, Missing data values Sorting and Merging Data Frames, Applying Functions, and Visualizing Dataframes.***

Table of Content

What is Data Processing in Python?

Data processing in Python refers to manipulating, transforming, and analyzing data by using Python. It contains a series of operations that aim to change raw data into structured data. or meaningful insights. By converting raw data into meaningful insights it makes it suitable for analysis, visualization, or other applications.Python provides several libraries and tools that facilitate efficient data processing, making it a popular choice for working with diverse datasets.
Refer to this article - Introduction to Data Processing

What is Pandas?
Pandas is a powerful, fast, and open-source library built on NumPy. It is used for data manipulation and real-world data analysis in Python. Easy handling of missing data, Flexible reshaping and pivoting of data sets, and size mutability make pandas a great tool for performing data manipulation and handling the data efficiently.

Loading Data in Pandas DataFrame
Reading CSV file using **pd.read_csv** and loading data into a data frame. Import pandas as using pd for the shorthand. You can download the data from here.

```
#Importing pandas library
import pandas as pd

#Loading data into a DataFrame
data_frame=pd.read_csv('Mall_Customers.csv')
```

Printing rows of the Data
By default, **data_frame.head()** displays the first five rows and **data_frame.tail()** displays last five rows. If we want to get first 'n' number of rows then we use, **data_frame.head(n)** similar is the syntax to print the last n rows of the data frame.

```
#displaying first five rows
display(data_frame.head())

#displaying last five rows
```

```
display(data_frame.tail())
```

Output:

```
  CustomerID  Genre  Age  Annual Income (k$)  Spending Score (1-100)
0          1   Male   19                  15                      39
1          2   Male   21                  15                      81
2          3 Female   20                  16                       6
3          4 Female   23                  16                      77
4          5 Female   31                  17                      40
     CustomerID  Genre  Age  Annual Income (k$)  Spending Score (1-100)
195         196 Female   35                 120                      79
196         197 Female   45                 126                      28
197         198   Male   32                 126                      74
198         199   Male   32                 137                      18
199         200   Male   30                 137                      83
[4]
0s
# Program to print all the column name of the dataframe
print(list(data_frame.columns))
```

Printing the column names of the DataFrame

```
# Program to print all the column name of the dataframe
print(list(data_frame.columns))
```

Output:

```
['CustomerID', 'Genre', 'Age', 'Annual Income (k$)', 'Spending Score (1-100)']
```

Summary of Data Frame

The functions **info()** prints the summary of a DataFrame that includes the data type of each column, RangeIndex (number of rows), columns, non-null values, and memory usage.

```
data_frame.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype 
---  ------                  --------------  ----- 
 0   CustomerID              200 non-null    int64 
 1   Genre                   200 non-null    object
 2   Age                     200 non-null    int64 
 3   Annual Income (k$)      200 non-null    int64 
 4   Spending Score (1-100)  200 non-null    int64 
dtypes: int64(4), object(1)
memory usage: 7.9+ KB
```

Descriptive Statistical Measures of a DataFrame

The **describe()** function outputs descriptive statistics which include those that summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN values. For numeric data, the result's index will include count, mean, std, min, and max as well as lower, 50, and upper percentiles. For object data (e.g. strings), the result's index will include count, unique, top, and freq.

```
data_frame.describe()
```

Output:

```
       CustomerID         Age  Annual Income (k$)  Spending Score (1-100)
count  200.000000  200.000000          200.000000              200.000000
mean   100.500000   38.850000           60.560000               50.200000
std     57.879185   13.969007           26.264721               25.823522
min      1.000000   18.000000           15.000000                1.000000
25%     50.750000   28.750000           41.500000               34.750000
50%    100.500000   36.000000           61.500000               50.000000
```

```
75%     150.250000   49.000000          78.000000          73.000000
max     200.000000   70.000000          137.000000         99.000000
```

Missing Data Handing

Find missing values in the dataset

The **isnull( )** detects the missing values and returns a boolean object indicating if the values are NA. The values which are none or empty get mapped to true values and not null values get mapped to false values.

```
data_frame.isnull( )
```

Output:

```
   CustomerID   Genre    Age    Annual Income (k$)   Spending Score (1-100)
0       False   False  False                False                   False
1       False   False  False                False                   False
2       False   False  False                False                   False
3       False   False  False                False                   False
4       False   False  False                False                   False
..        ...     ...    ...                  ...                     ...
195     False   False  False                False                   False
196     False   False  False                False                   False
197     False   False  False                False                   False
198     False   False  False                False                   False
199     False   False  False                False                   False
[200 rows x 5 columns]
[8]
0s
```

Find the number of missing values in the dataset

To find out the number of missing values in the dataset, use data_frame.isnull( ).sum( ). In the below example, the dataset doesn't contain any null values. Hence, each column's output is 0.

```
data_frame.isnull().sum()
```

Output:

```
CustomerID                0
Genre                     0
Age                       0
Annual Income (k$)        0
Spending Score (1-100)    0
dtype: int64
```

Removing missing values

The **data_frame.dropna( )** function removes columns or rows which contains atleast one missing values.

```
data_frame = data_frame.dropna()
```

By default, **data_frame.dropna( )** drops the rows where at least one element is missing. **data_frame.dropna(axis = 1)** drops the columns where at least one element is missing.

Fill in missing values

We can fill null values using **data_frame.fillna( )** function.

```
 data_frame = data_frame.fillna(value)
```

But by using the above format all the null values will get filled with the same values. To fill different values in the different columns we can use.

```
data_frame[col] = data_frame[col].fillna(value)
```

Row and column manipulations

Removing rows

By using the **drop(index)** function we can drop the row at a particular index. If we want to replace the data_frame with the row removed then add **inplace = True** in the drop function.

```
#Removing 4th indexed value from the dataframe
data_frame.drop(4).head()
```

Output:

CustomerID		Genre	Age	Annual	Income	(k$)	Spending	Score	(1-100)
0	1	Male	19			15			39
1	2	Male	21			15			81
2	3	Female	20			16			6
3	4	Female	23			16			77
5	6	Female	22			17			76

[ ]

This function can also be used to remove the columns of a data frame by adding the attribute **axis =1** and providing the list of columns we would like to remove.

Renaming rows

The rename function can be used to rename the rows or columns of the data frame.

```
data_frame.rename({0:"First",1:"Second"})
```

Output:

CustomerID		Genre	Age	Annual	Income	(k$)	Spending	Score	(1-100)
First	1	Male	19			15			39
Second	2	Male	21			15			81
2	3	Female	20			16			6
3	4	Female	23			16			77
4	5	Female	31			17			40
...	...	...	...			...			...
195	196	Female	35			120			79
196	197	Female	45			126			28
197	198	Male	32			126			74
198	199	Male	32			137			18
199	200	Male	30			137			83

[200 rows x 5 columns]

Adding new columns

```
#Creates a new column with all the values equal to 1
data_frame['NewColumn'] = 1
data_frame.head()
```

Output:

CustomerID		Genre	Age	Annual	Income	(k$)	Spending	Score	(1-100)	\
0	1	Male	19			15			39	
1	2	Male	21			15			81	
2	3	Female	20			16			6	
3	4	Female	23			16			77	
4	5	Female	31			17			40	
NewColumn										
0										1
1										1
2										1
3										1
4	1									

Sorting DataFrame values

Sort by column

The **sort_values( )** are the values of the column whose name is passed in the **by** attribute in the ascending order by default we can set this attribute to false to sort the array in the descending order.

```
data_frame.sort_values(by='Age', ascending=False).head()
```

Output:

CustomerID		Genre	Age	Annual	Income	(k$)	Spending	Score	(1-100)	\
70	71	Male	70			49			55	
60	61	Male	70			46			56	
57	58	Male	69			44			46	
90	91	Female	68			59			55	

67	68	Female	68	48	48

NewColumn

70	1
60	1
57	1
90	1
67	1

Sort by multiple columns

```
data_frame.sort_values(by=['Age','Annual Income (k$)']).head(10)
```

Output:

CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)	\
33	34 Male	18	33		92
65	66 Male	18	48		59
91	92 Male	18	59		41
114	115 Female	18	65		48
0	1 Male	19	15		39
61	62 Male	19	46		55
68	69 Male	19	48		59
111	112 Female	19	63		54
113	114 Male	19	64		46
115	116 Female	19	65		50

NewColumn

33	1
65	1
91	1
114	1
0	1
61	1
68	1
111	1
113	1
115	1

Merge Data Frames

The **merge()** function in pandas is used for all standard database join operations. Merge operation on data frames will join two data frames based on their common column values. Let's create a data frame.

```
#Creating dataframe1
df1 = pd.DataFrame({
    'Name':['Jeevan', 'Raavan', 'Geeta', 'Bheem'],
    'Age':[25, 24, 52, 40],
    'Qualification':['Msc', 'MA', 'MCA', 'Phd']})
df1
```

Output:

	Name	Age	Qualification
0	Jeevan	25	Msc
1	Raavan	24	MA
2	Geeta	52	MCA
3	Bheem	40	Phd

Now we will create another data frame.

```
#Creating dataframe2
df2 = pd.DataFrame({'Name':['Jeevan', 'Raavan', 'Geeta', 'Bheem'],
                    'Salary':[100000, 50000, 20000, 40000]})
df2
```

Output:

	Name	Salary
0	Jeevan	100000
1	Raavan	50000

```
2                                    Geeta                                  20000
3  Bheem  40000
```

Now. let's merge these two data frames created earlier.

```
#Merging two dataframes
df = pd.merge(df1, df2)
df
```

Output:

```
   Name                     Age           Qualification            Salary
0        Jeevan          25                                 Msc       100000
1       Raavan         24                                 MA         50000
2           Geeta          52                               MCA        20000
3  Bheem  40     Phd  40000
```

Apply Function

By defining a function beforehand

The apply( ) function is used to iterate over a data frame. It can also be used with lambda functions.

```
# Apply function
def fun(value):
  if value > 70:
    return "Yes"
  else:
    return "No"

data_frame['Customer Satisfaction'] = data_frame['Spending Score (1-100)'].apply(fun)
data_frame.head(10)
```

Output:

```
  CustomerID          Genre     Age     Annual   Income   (k$)    Spending   Score   (1-100)   \
0                 1      Male      19                           15                                39
1                 2      Male      21                           15                                81
2                 3   Female     20                           16                                 6
3                 4   Female     23                           16                                77
4                 5   Female     31                           17                                40
5                 6   Female     22                           17                                76
6                 7   Female     35                           18                                 6
7                 8   Female     23                           18                                94
8                 9      Male      64                           19                                 3
9                10   Female     30                           19                                72
  NewColumn                         Customer                        Satisfaction
0                          1                                                   No
1                           1                                                  Yes
2                          1                                                   No
3                           1                                                  Yes
4                          1                                                   No
5                           1                                                  Yes
6                          1                                                   No
7                           1                                                  Yes
8                          1                                                   No
9      1         Yes
```

By using the lambda operator

This syntax is generally used to apply log transformations and normalize the data to bring it in the range of 0 to 1 for particular columns of the data.

```
const = data_frame['Age'].max()
data_frame['Age'] = data_frame['Age'].apply(lambda x: x/const)
data_frame.head()
```

Output:

	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)	\
0	1	Male	0.271429	15	39	
1	2	Male	0.300000	15	81	
2	3	Female	0.285714	16	6	
3	4	Female	0.328571	16	77	
4	5	Female	0.442857	17	40	

	NewColumn		Customer	Satisfaction
0		1		No
1		1		Yes
2		1		No
3		1		Yes
4	1	No		

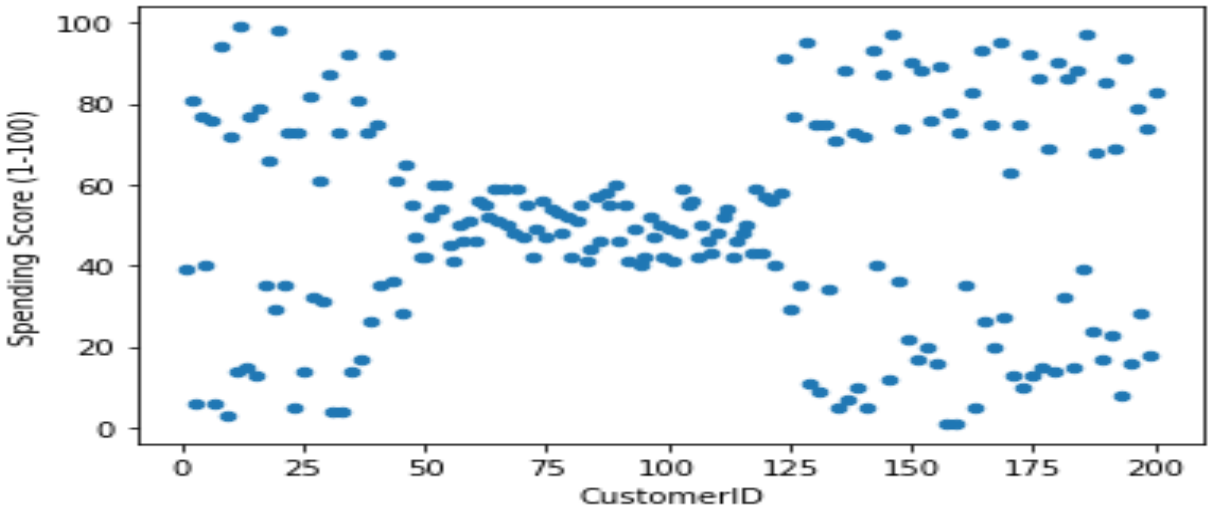
Visualizing DataFrame

Scatter plot

The **plot( )** function is used to make plots of the data frames.

```
# Visualization
data_frame.plot(x ='CustomerID', y='Spending Score (1-100)',kind = 'scatter')
```

Output:

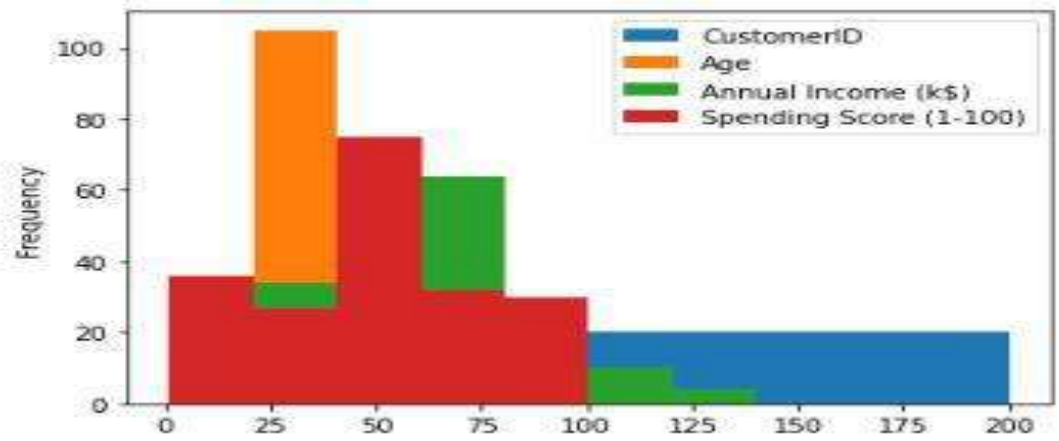


Histogram

The **plot.hist( )** function is used to make plots of the data frames.

```
data_frame.plot.hist()
```

Output:



Data Normalization with Pandas

Data normalization is the process of scaling numeric features to a standard range, preventing large values from dominating the learning process in machine learning models. It is a important step in machine learning and data analysis ensure that numerical features are on a similar scale for optimal model performance.

Normalization helps to improve algorithm performance particularly for distance-based models like **K-Nearest Neighbors (KNN)** and **Support Vector Machines (SVM).** It is important because:

- Avoids numerical instability in models
- Speeds up convergence in gradient-based algorithms
- Ensures all features contribute equally to the analysis

Steps for Data Normalization in Pandas

Here we will apply some techniques to normalize the data and discuss these with the help of examples. For this let's understand the steps needed for data normalization with Pandas.

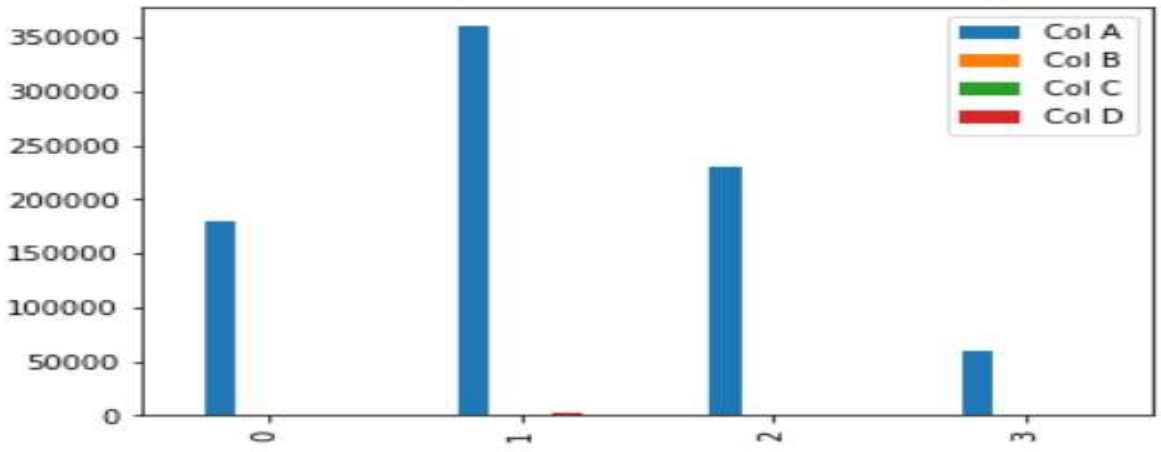
1. Import the required libraries
2. Load or create a dataset
3. Apply different normalization techniques
4. Visualize the results

Let's create a sample dataset using **Pandas** and visualize it.

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame([
   [180000, 110, 18.9, 1400],
   [360000, 905, 23.4, 1800],
   [230000, 230, 14.0, 1300],
   [60000, 450, 13.5, 1500]
], columns=['Col A', 'Col B', 'Col C', 'Col D'])

print(df)

df.plot(kind='bar')
plt.show()
```

Output:

	Col A	Col B	Col C	Col D
0	180000	110	18.9	1400
1	360000	905	23.4	1800
2	230000	230	14.0	1300
3	60000	450	13.5	1500



Normalization Techniques in Pandas

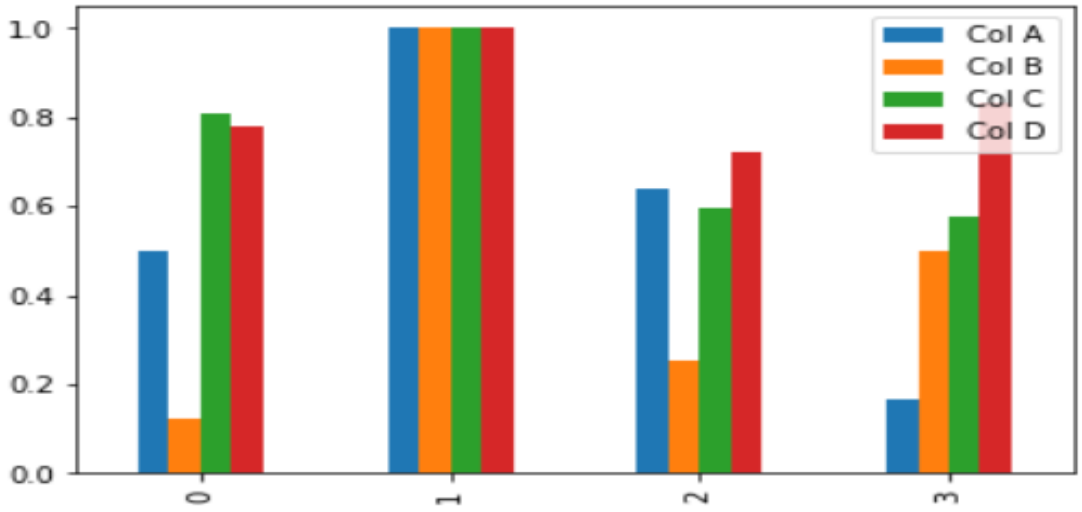
1. Maximum Absolute Scaling

This technique rescales each feature between **-1 and 1** by dividing all values by the **maximum absolute value** in that column. This technique is especially useful when your data **doesn't contain negative numbers** and you want to preserve the data's sparsity. We can apply the maximum absolute scaling in Pandas using the .max() and .abs() methods as shown below. Let's apply normalization techniques one by one.

```
max_scaled = df.copy()

for column in df_max_scaled.columns:
  max_scaled[column] = max_scaled[column] / max_scaled[column].abs().max()

print(max_scaled)

max_scaled.plot(kind='bar')
plt.show()
```

Output :

	Col A	Col B	Col C	Col D
0	0.500000	0.121547	0.807692	0.777778
1	1.000000	1.000000	1.000000	1.000000
2	0.638889	0.254144	0.598291	0.722222
3	0.166667	0.497238	0.576923	0.833333



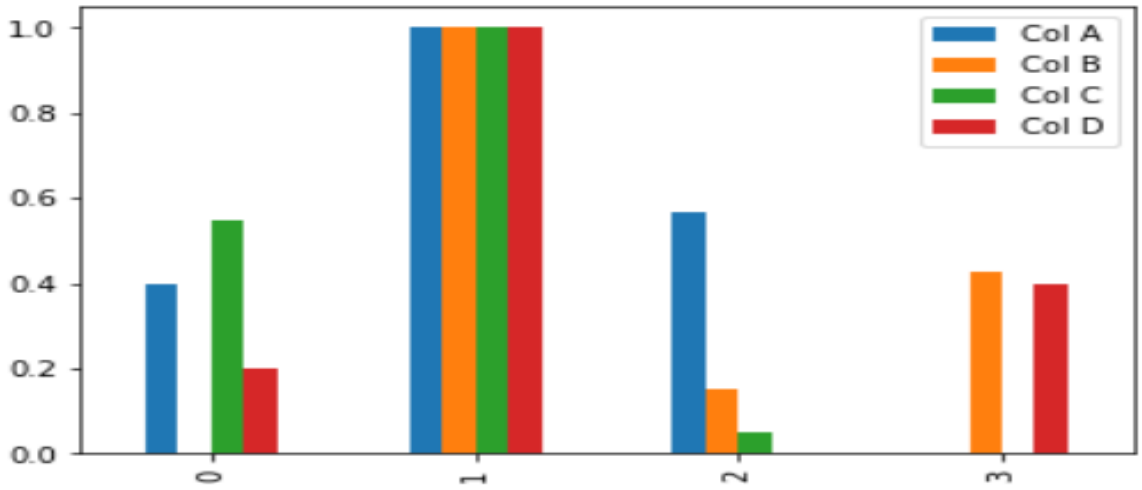
2. The min-max feature scaling

The min-max approach also called normalization rescales the feature to a hard and fast range of [0,1] by subtracting the minimum value of the feature then dividing by the range. . It works well for models like K-Nearest Neighbors (KNN) which compare distance between data points. We can apply the min-max scaling in Pandas using the .min() and .max() methods.

```
scaled = df.copy()

for column in df_min_max_scaled.columns:
  scaled[column] = (scaled[column] - scaled[column].min()) / (scaled[column].max() - scaled[column].min())

print(scaled)
scaled.plot(kind='bar')
plt.show()
```

Output :

	Col A	Col B	Col C	Col D
0	0.400000	0.000000	0.545455	0.2
1	1.000000	1.000000	1.000000	1.0
2	0.566667	0.150943	0.050505	0.0
3	0.000000	0.427673	0.000000	0.4



After scaling **the smallest value becomes 0** and **the largest becomes 1**. All other values lie between these two. This makes it easier for the machine learning model to handle features fairly.

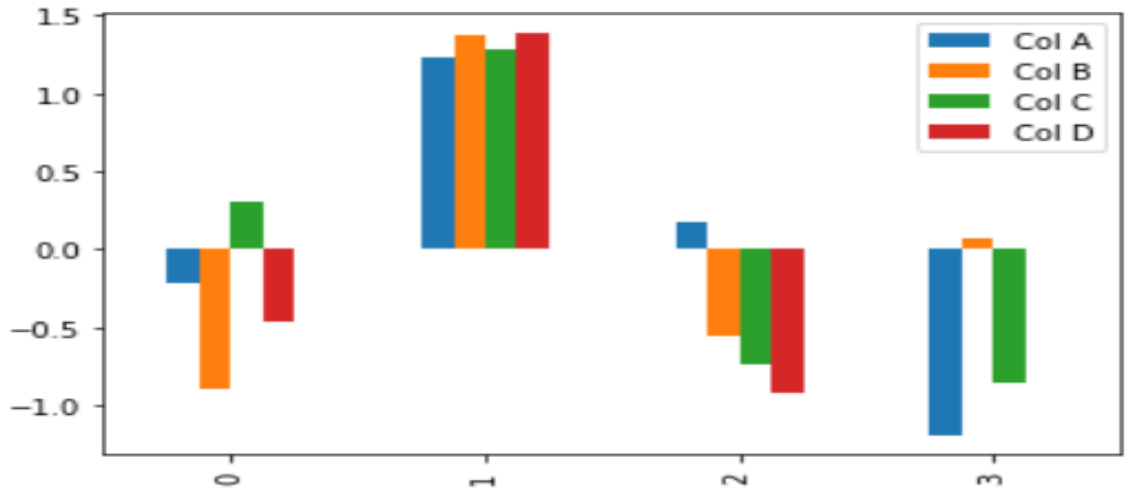
3. The z-score method

The z-score method often called standardization changes the values in each column so that they have a **mean** of 0 and a **standard deviation** of 1. This technique is best when your data follow a **normal distribution** or when you want to treat values in terms of how far they are from the average.

```
z_scaled = df.copy()

for column in z_scaled.columns:
  z_scaled[column] = (z_scaled[column] - z_scaled[column].mean()) / z_scaled[column].std()

print(z_scaled)

z_scaled.plot(kind='bar')
plt.show()
```

Output :

	Col A	Col B	Col C	Col D
0	-0.221422	-0.895492	0.311486	-0.46291
1	1.227884	1.373564	1.278167	1.38873
2	0.181163	-0.552993	-0.741122	-0.92582
3	-1.187625	0.074922	-0.848531	0.00000



Data Manipulation in Python using Pandas

In Machine Learning, the model requires a dataset to operate, i.e. to train and test. But data doesn't come fully prepared and ready to use. There are discrepancies like Nan/ Null / NA values in many rows and columns. Sometimes the data set also contains some of the rows and columns which are not even required in the operation of our model. In such conditions, it requires proper cleaning and modification of the data set to make it an efficient input for our model. We achieve that by practicing **Data Wrangling** before giving data input to the model.
Today, we will get to know some methods using Pandas which is a famous library of Python. And by using it we can make out data ready to use for training the model and hence getting some useful insights from the results.

Installing Pandas
Before moving forward, ensure that Pandas is installed in your system. If not, you can use the following command to install it:

```
pip install pandas
```

Creating DataFrame
Let's dive into the programming part. Our first aim is to create a Pandas dataframe in Python, as you may know, pandas is one of the most used libraries of Python.
Code:

```
# Importing the pandas library
import pandas as pd

# creating a dataframe object
student_register = pd.DataFrame()

# assigning values to the
# rows and columns of the dataframe
student_register['Name'] = ['Abhijit','Smriti',
                'Akash', 'Roshni']
```

```
student_register['Age'] = [20, 19, 20, 14]
student_register['Student'] = [False, True,
                    True, False]

print(student_register)
```

Output:

```
    Name  Age  Student
0  Abhijit   20   False
1   Smriti   19    True
2    Akash   20    True
3   Roshni   14   False
```

As you can see, the dataframe object has four rows [0, 1, 2, 3] and three columns[“Name”, “Age”, “Student”] respectively. The column which contains the index values i.e. [0, 1, 2, 3] is known as the **index column**, which is a default part in pandas datagram. We can change that as per our requirement too because Python is powerful.

Adding data in DataFrame using Append Function

Next, for some reason we want to add a new student in the datagram, i.e you want to add a new row to your existing data frame, that can be achieved by the following code snippet.
One important concept is that the “dataframe” object of Python, consists of rows which are “series” objects instead, stack together to form a table. Hence adding a new row means creating a new series object and appending it to the dataframe.

Code:

```
# creating a new pandas
# series object
new_person = pd.Series(['Mansi', 19, True],
                index = ['Name', 'Age',
                     'Student'])

# using the .append() function
# to add that row to the dataframe
student_register.append(new_person, ignore_index = True)
print(student_register)
```

Output:

```
    Name  Age  Student
0  Abhijit   20   False
1   Smriti   19    True
2    Akash   20    True
3   Roshni   14   False
```

Data Manipulation on Dataset

Till now, we got the gist of how we can create dataframe, and add data to it. But how will we perform these operations on a big dataset. For this let's take a new dataset

Getting Shape and information of the data

Let's exact information of each column, i.e. what type of value it stores and how many of them are unique. There are three support functions, **.shape**, **.info**() and **.corr**() which output the shape of the table, information on rows and columns, and correlation between numerical columns.

Code:

```
# dimension of the dataframe
print('Shape: ')
print(student_register.shape)
print('-------------------------------------')
# showing info about the data
print('Info: ')
print(student_register.info())
```

```
print('-------------------------------------')
# correlation between columns
print('Correlation: ')
print(student_register.corr())
```

Output:

```
Shape:
(4, 3)
-------------------------------------
Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Name     4 non-null      object
 1   Age      4 non-null      int64
 2   Student  4 non-null      bool
dtypes: bool(1), int64(1), object(1)
memory usage: 196.0+ bytes
None
-------------------------------------
Correlation:
             Age   Student
Age      1.000000  0.502519
Student  0.502519  1.000000
```

In the above example, the **.shape** function gives an output (4, 3) as that is the size of the created dataframe.

The description of the output given by .info() method is as follows:

1. **RangeIndex** describes about the index column, i.e. [0, 1, 2, 3] in our datagram. Which is the number of rows in our dataframe.
2. As the name suggests **Data columns** give the total number of columns as output.
3. **Name, Age, Student** are the name of the columns in our data, non-null tells us that in the corresponding column, there is no NA/ Nan/ None value exists. **object**, **int64** and **bool** are the datatypes each column have.
4. **dtype** gives you an overview of how many data types present in the datagram, which in term simplifies the data cleaning process.
   Also, in high-end machine learning models, **memory usage** is an important term, we can't neglect that.

Getting Statistical Analysis of Data

Before processing and wrangling any data you need to get the total overview of it, which includes statistical conclusions like **standard deviation(std), mean and it's quartile distributions**.

```
# for showing the statistical
# info of the dataframe
print('Describe')
print(student_register.describe())
```

Output:

```
Describe
             Age
count   4.000000
mean   18.250000
std     2.872281
min    14.000000
25%    17.750000
50%    19.500000
75%    20.000000
max    20.000000
```

The description of the output given by .describe() method is as follows:

1. **count** is the number of rows in the dataframe.
2. **mean** is the mean value of all the entries in the “Age” column.
3. **std** is the standard deviation of the corresponding column.
4. **min** and max are the minimum and maximum entry in the column respectively.
5. 25%, 50% and 75% are the **First Quartiles**, **Second Quartile(Median)** and **Third Quartile** respectively, which gives us important info on the distribution of the dataset and makes it simpler to apply an ML model.

Dropping Columns from Data

Let's drop a column from the data. We will use the drop function from the pandas. We will keep axis = 1 for columns.

```
students = student_register.drop('Age', axis=1)
print(students.head())
```

Output:

	Name	Student
0	Abhijit	False
1	Smriti	True
2	Akash	True
3	Roshni	False

From the output, we can see that the 'Age' column is dropped.

Dropping Rows from Data

Let's try dropping a row from the dataset, for this, we will use drop function. We will keep axis=0.

```
students = students.drop(2, axis=0)
print(students.head())
```

Output:

	Name	Student
0	Abhijit	False
1	Smriti	True
3	Roshni	False

Pandas dataframe.groupby() Method

Pandas groupby() function is a powerful tool used to split a DataFrame into groups based on one or more columns, allowing for efficient data analysis and aggregation. It follows a "split-apply-combine" strategy, where data is divided into groups, a function is applied to each group, and the results are combined into a new DataFrame. **For example, if you have a dataset of sales transactions, you can use groupby() to group the data by product category and calculate the total sales for each category.**

```
import pandas as pd
data = {'Product_Category': ['Electronics', 'Furniture', 'Electronics', 'Furniture'],'Sales': [200, 150, 300, 100]}
df = pd.DataFrame(data)

# Group by Product_Category and sum Sales
result = df.groupby('Product_Category')['Sales'].sum()
display(result)
```

Product_Category	Sales
Electronics	500
Furniture	250

dtype: int64

The code is providing total sales for each product category, demonstrating the core idea of grouping data and applying an aggregation function.

How to Use Pandas GroupBy Method?

The groupby() function in Pandas involves three main steps: **Splitting**, **Applying**, and **Combining**.

- **Splitting**: This step involves dividing the DataFrame into groups based on some criteria. The groups are defined by unique values in one or more columns.
- **Applying**: In this step, a function is applied to each group independently. You can apply various functions to each group, such as:
    - **Aggregation:** Calculate summary statistics (e.g., sum, mean, count) for each group.
    - **Transformation:** Modify the values within each group.
    - **Filtering:** Keep or discard groups based on certain conditions.
- **Combining**: Finally, the results of the applied function are combined into a new DataFrame or Series.

The groupby method has several parameters that can be customized:
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, observed=False, dropna=True)

Parameters :

- ***by****: Required parameter to specify the column(s) to group by.*
- ***axis****: Optional, specifies the axis to group by (default is 0 for rows).*
- ***level****: Optional, used for grouping by a certain level in a MultiIndex.*
- ***as_index****: Optional, whether to use the group labels as the index (default is True).*
- ***sort****: Optional, whether to sort the group keys (default is True).*
- ***group_keys****: Optional, whether to add the group keys to the index (default is True).*
- ***dropna****: Optional, whether to include rows/columns with NULL values (default is True*

Example 1: Grouping by a Single Column

In this example, we will demonstrate how to group data by a single column using the groupby method. We will work with NBA-dataset that contains information about NBA players, including their teams, points scored, and assists. We'll group the data by the Team column and calculate the total points scored for each team.

```
import pandas as pd
df = pd.read_csv("https://media.TechforTech.org/wp-content/uploads/nba.csv")

team = df.groupby('Team')
print(team.first()) # Let's print the first entries in all the groups formed.
```

Output:

Team	Name	Number	Position	Age	Height	Weight	College	Salary
Atlanta Hawks	Kent Bazemore	24.0	SF	26.0	6-5	201.0	Old Dominion	2000000.0
Boston Celtics	Avery Bradley	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
Brooklyn Nets	Bojan Bogdanovic	44.0	SG	27.0	6-8	216.0	Oklahoma State	3425510.0
Charlotte Hornets	Nicolas Batum	5.0	SG	27.0	6-8	200.0	Virginia Commonwealth	13125306.0
Chicago Bulls	Cameron Bairstow	41.0	PF	25.0	6-9	250.0	New Mexico	845059.0
Cleveland Cavaliers	Matthew Dellavedova	8.0	PG	25.0	6-4	198.0	Saint Mary's	1147276.0

Grouping by Multiple Columns

Grouping by multiple columns allows you to break down the data into finer categories and compute statistics for each unique combination of those columns. Let's use the same NBA dataset and group the data by both Team and positionto calculate the total points scored by each position within each team.

```
import pandas as pd
df = pd.read_csv("https://media.TechforTech.org/wp-content/uploads/nba.csv")

grouping = df.groupby(['Team', 'Position'])
print(grouping.first())
```

Output:

Team	Position	Name	Number	Age	Height	Weight	College	Salary
Atlanta Hawks	**C**	Al Horford	15.0	30.0	6-10	245.0	Florida	12000000.0
	PF	Kris Humphries	43.0	31.0	6-9	235.0	Minnesota	1000000.0
	PG	Dennis Schroder	17.0	22.0	6-1	172.0	Wake Forest	1763400.0
	SF	Kent Bazemore	24.0	26.0	6-5	201.0	Old Dominion	2000000.0
	SG	Tim Hardaway Jr.	10.0	24.0	6-6	205.0	Michigan	1304520.0
...	...	...	...	...	...	...	...	...

Applying Aggregation with GroupBy

Aggregation is one of the most common operations when using groupby. After grouping the data, you can apply functions like sum(), mean(), min(), max(), and more.

- **sum()**: Calculates the total sum of values for each group. Useful when you need to know the total amount of a numeric column grouped by specific categories.
- **mean()**: Computes the average value of each group, helpful for understanding trends and patterns within grouped data.
- **count()**: Counts the number of entries in each group, returns the number of non-null entries for each group.

Let's continue with the same NBA dataset and demonstrate how aggregation works in practice using the sum(), mean(), and count() functions. The example will group the data by both Team and Position, and apply all three aggregation functions to **understand the total salary, average salary, and the number of players in each group.**

```
import pandas as pd
df = pd.read_csv("https://media.TechforTech.org/wp-content/uploads/nba.csv")

aggregated_data = df.groupby(['Team', 'Position']).agg(
  total_salary=('Salary', 'sum'),
  avg_salary=('Salary', 'mean'),
  player_count=('Name', 'count')
)

print(aggregated_data)
```

Output:

Team	Position	total_salary	avg_salary	player_count
Atlanta Hawks	**C**	22756250.0	7.585417e+06	3
	PF	23952268.0	5.988067e+06	4
	PG	9763400.0	4.881700e+06	2
	SF	6000000.0	3.000000e+06	2
	SG	10431032.0	2.607758e+06	4
...	...	...	...	...
Washington Wizards	**C**	24490429.0	8.163476e+06	3
	PF	11300000.0	5.650000e+06	2
	PG	18022415.0	9.011208e+06	2
	SF	11158800.0	2.789700e+06	4
	SG	11356992.0	2.839248e+06	4

How to Apply Transformation Methods?

Transformation functions return an object that is indexed the same as the original group. This is useful when you need to apply operations that maintain the original structure of the data, such as normalization or standardization within groups. **Purpose: Apply group-specific operations while maintaining the original shape of the dataset.** Unlike aggregation, which reduces data, transformations allow group-specific modifications without altering the shape of the data.

For example: Let's understand how to: Rank players within their teams based on their salaries - Ranking players within their teams by salary can help identify the highest- and lowest-paid players in each group.

```
import pandas as pd
df = pd.read_csv("https://media.TechforTech.org/wp-content/uploads/nba.csv")

# Rank players within each team by Salary
df['Rank within Team'] = df.groupby('Team')['Salary'].transform(lambda x: x.rank(ascending=False))
print(df)
```

Output:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary	Rank within Team
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0	2.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0	4.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0	14.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0	5.0
...	...	...	...	...	...	...	...	...	...	...
453	Shelvin Mack	Utah Jazz	8.0	PG	26.0	6-3	203.0	Butler	2433333.0	8.0
454	Raul Neto	Utah Jazz	25.0	PG	24.0	6-1	179.0	NaN	900000.0	15.0
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0	6.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0	14.0
457	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

458 rows × 10 columns

Grouping and Aggregating with Pandas

Aggregation in Pandas

Aggregation means applying a mathematical function to summarize data. It can be used to get a summary of columns in our dataset like getting sum, minimum, maximum etc. from a particular column of our dataset. The function used for aggregation is **agg()** the parameter is the function we want to perform. Some functions used in the aggregation are:

Function	Description
sum()	Compute sum of column values
min()	Compute min of column values
max()	Compute max of column values
mean()	Compute mean of column
size()	Compute column sizes
describe()	Generates descriptive statistics
first()	Compute first of group values
last()	Compute last of group values
count()	Compute count of column values
std()	Standard deviation of column
var()	Compute variance of column

sem()	Standard error of the mean of column

Creating a Sample Dataset

Let's create a small dataset of student marks in Maths, English, Science and History.

```
import pandas as pd

df = pd.DataFrame([[9, 4, 8, 9],
                   [8, 10, 7, 6],
                   [7, 6, 8, 5]],
                  columns=['Maths',  'English',
                           'Science', 'History'])

print(df)
```

Output:

	Maths	English	Science	History
0	9	4	8	9
1	8	10	7	6
2	7	6	8	5

Now that we have a dataset let's perform aggregation.

1. Summing Up All Values (sum())

The sum() function adds up all values in each column.

```
df.sum()
```

Output:

```
Maths      24
English    20
Science    23
History    20
dtype: int64
```

2. Getting a Summary (describe())

Instead of calculating sum, mean, min and max separately we can use describe() which provides all important statistics in one go.

```
df.describe()
```

Output:

	Maths	English	Science	History
count	3.0	3.000000	3.000000	3.000000
mean	8.0	6.666667	7.666667	6.666667
std	1.0	3.055050	0.577350	2.081666
min	7.0	4.000000	7.000000	5.000000
25%	7.5	5.000000	7.500000	5.500000
50%	8.0	6.000000	8.000000	6.000000
75%	8.5	8.000000	8.000000	7.500000
max	9.0	10.000000	8.000000	9.000000

3. Applying Multiple Aggregations at Once (agg())
The .agg() function lets you apply multiple aggregation functions at the same time.

```
df.agg(['sum', 'min', 'max'])
```

Output:

	Maths	English	Science	History
sum	24	20	23	20
min	7	4	7	5
max	9	10	8	9

Grouping in Pandas
Grouping in Pandas means organizing your data into groups based on some columns. Once grouped you can perform actions like finding the total, average, count or even pick the first row from each group. This method follows a **split-apply-combine** process:

- **Split** the data into groups
- **Apply** some calculation like sum, average etc.
- **Combine** the results into a new table.

Let's understand grouping in Pandas using a small bakery order dataset as an example.

```
import pandas as pd

data = {
  'Item': ['Cake', 'Cake', 'Bread', 'Pastry', 'Cake'],
  'Flavor': ['Chocolate', 'Vanilla', 'Whole Wheat', 'Strawberry', 'Chocolate'],
  'Price': [250, 220, 80, 120, 250]
}

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

Output:

```
      Item        Flavor  Price
0     Cake     Chocolate    250
1     Cake       Vanilla    220
2    Bread  Whole Wheat      80
3  Pastry    Strawberry    120
4     Cake     Chocolate    250
```

1. Grouping Data by One Column Using groupby()
Let's say we want to group the orders based on the Item column.

```
grouped = df.groupby('Item')
print(grouped)
```

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7867484be150>

This doesn't show the result directly it just creates a **grouped object**. To actually see the data we need to apply a method like .sum(), .mean() or first(). Let's find the **total price** of each item sold:

```
print(df.groupby('Item')['Price'].sum())
```

Output:

```
Item
Bread       80
Cake       720
Pastry     120
Name: Price, dtype: int64
```

2. Grouping by Multiple Columns

Now let's group by **Item and Flavor** to see how each flavored item sold.

```
print(df.groupby(['Item', 'Flavor'])['Price'].sum())
```

Output:

```
Item     Flavor
Bread    Whole Wheat     80
Cake     Chocolate      500
         Vanilla        220
Pastry   Strawberry     120
Name: Price, dtype: int64
```

Different Types of Joins in Pandas

The Pandas module contains various features to perform various operations on Dataframes like join, concatenate, delete, add, etc. In this article, we are going to discuss the various types of join operations that can be performed on Pandas Dataframe. There are five types of Joins in Pandas.

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join or simply Outer Join
- Index Join

To understand different types of joins, we will first make two DataFrames, namely **a** and **b**.

Dataframe a:

```
# importing pandas

import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
   'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# printing the dataframe
a
```

Output:

	id	val1
0	1	a
1	2	b
2	10	c
3	12	d

DataFrame b:

```
# importing pandas
import pandas as pd

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
    'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# printing the dataframe
b
```

Output:

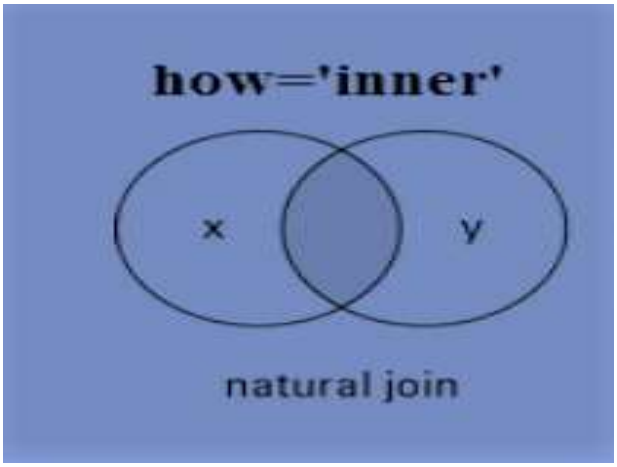
	id	val1
0	1	p
1	2	q
2	9	r
3	8	s

Types of Joins in Pandas

We will use these two Dataframes to understand the different types of joins.

Pandas Inner Join

Inner join is the most common type of join you'll be working with. It returns a Dataframe with only those rows that have common characteristics. This is similar to the intersection of two sets.



Example:

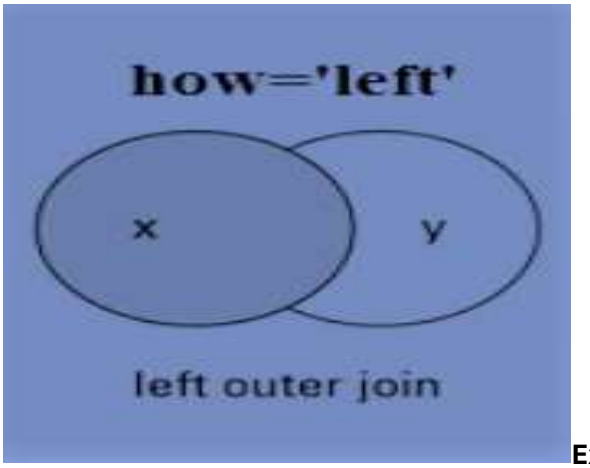
```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
    'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
    'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# inner join
df = pd.merge(a, b, on='id', how='inner')

# display dataframe
df
```

Output:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q

Pandas Left Join

With a left outer join, all the records from the first Dataframe will be displayed, irrespective of whether the keys in the first Dataframe can be found in the second Dataframe. Whereas, for the second Dataframe, only the records with the keys in the second Dataframe that can be found in the first Dataframe will be displayed.



Example:

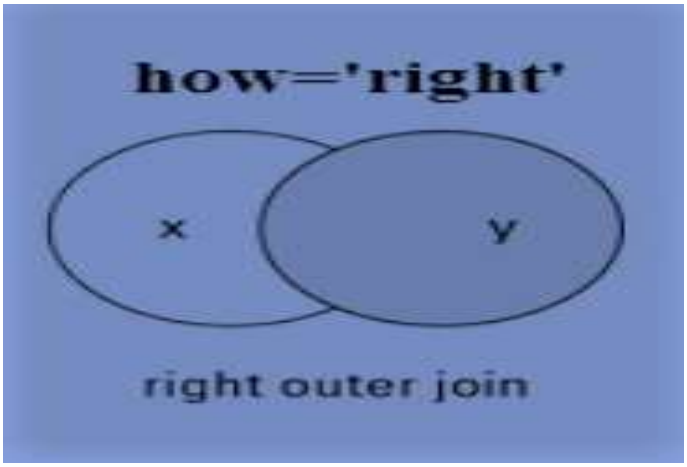
```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
    'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
    'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# left outer join
df = pd.merge(a, b, on='id', how='left')

# display dataframe
df
```

Output:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q
2	10	c	NaN
3	12	d	NaN

Pandas Right Outer Join

For a right join, all the records from the second Dataframe will be displayed. However, only the records with the keys in the first Dataframe that can be found in the second Dataframe will be displayed.



Example:

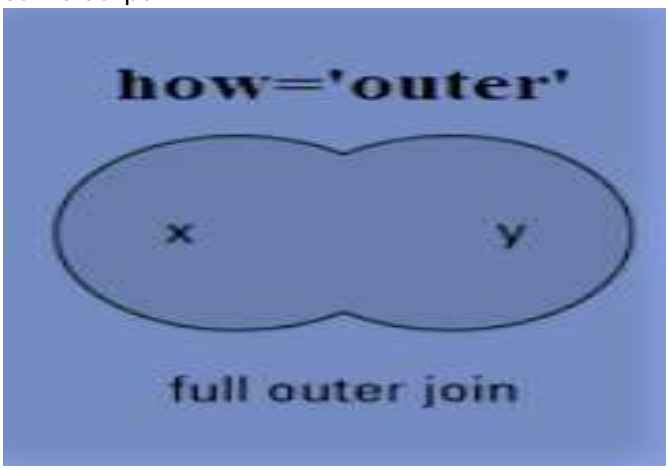
```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
    'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
    'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# right outer join
df = pd.merge(a, b, on='id', how='right')

# display dataframe
df
```

Output:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q
2	9	NaN	r
3	8	NaN	s

Pandas Full Outer Join

A full outer join returns all the rows from the left Dataframe, and all the rows from the right Dataframe, and matches up rows where possible, with NaNs elsewhere. But if the Dataframe is complete, then we get the same output.



Example:

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
    'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
    'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# full outer join
df = pd.merge(a, b, on='id', how='outer')

# display dataframe
df
```

Output:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q
2	10	c	NaN
3	12	d	NaN
4	9	NaN	r
5	8	NaN	s

Pandas Index Join

To merge the Dataframe on indices pass the *left_index* and *right_index* arguments as True i.e. both the Dataframes are merged on an index using default Inner Join.

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
   'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
   'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# index join
df = pd.merge(a, b, left_index=True, right_index=True)

# display dataframe
df
```

Data Visualization with Pandas

Pandas is a powerful open-source data analysis and manipulation library for Python. The library is particularly well-suited for handling labeled data such as tables with rows and columns. Pandas allows to create **various graphs directly from your data using built-in functions.**

This tutorial covers Pandas capabilities for visualizing data with line plots, area charts, bar plots, and more.

Key Features for Data Visualization with Pandas:

Pandas offers several features that make it a great choice for data visualization:

- **Variety of Plot Types**: Pandas supports various plot types including line plots, bar plots, histograms, box plots, and scatter plots.
- **Customization**: Users can customize plots by adding titles, labels, and styling enhancing the readability of the visualizations.
- **Handling of Missing Data**: Pandas efficiently handles missing data ensuring that visualizations accurately represent the dataset without errors.
- **Integration with Matplotlib**: Pandas integrates with Matplotlib that allow users to create a wide range of static, animated, and interactive plots.

Installation of Pandas

To get started you need to install Pandas using pip:

pip install pandas

Importing necessary libraries and data files

Once Pandas is installed, import the required libraries and load your data Sample CSV files **df1** and **df2** used in this tutorial can be downloaded from here.

```
import numpy as np
```

```
import pandas as pd

df1 = pd.read_csv('df1', index_col=0)
df2 = pd.read_csv('df2')
```

Explanation:

- **pd.read_csv('df1', index_col=0)** loads **df1.csv** and sets the first column as the index.
- **pd.read_csv('df2')** loads **df2.csv** with default indexing.

Pandas DataFrame Plots

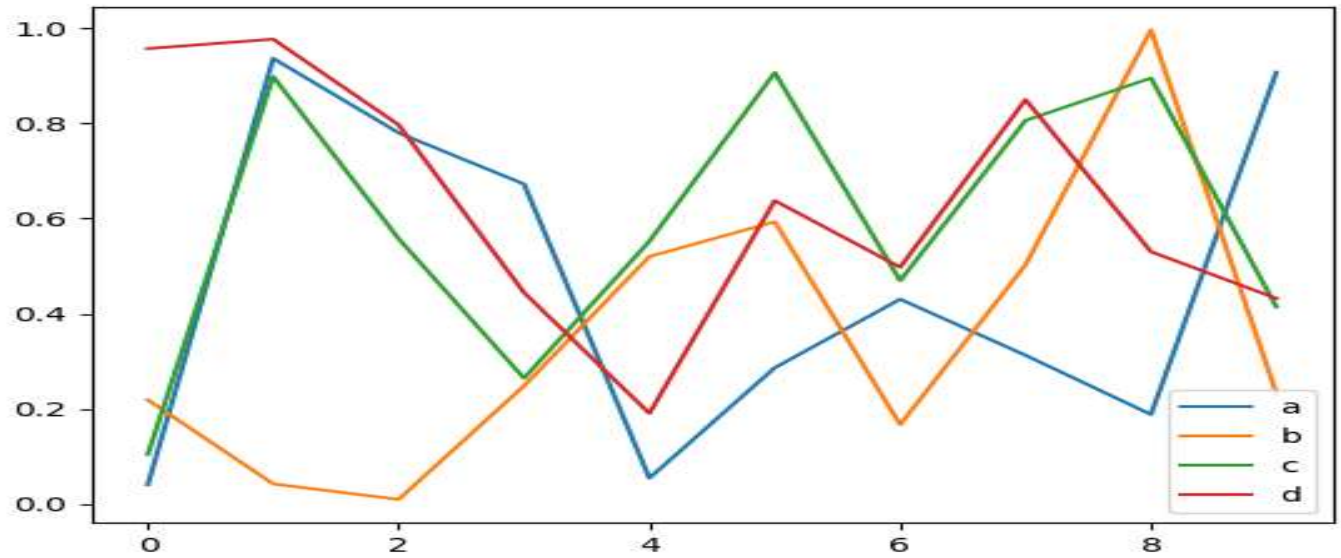
Pandas provides several built-in plotting functions to create various types of charts mainly focused on statistical data. These plots help visualize trends, distributions, and relationships within the data. Let's go through them one by one:

1. Line Plots using Pandas DataFrame

A Line plot is a graph that shows the frequency of data along a number line. It is best to use a line plot when the data is time series. It can be created using **Dataframe.plot()** function.

```
df2.plot()
```

Output:



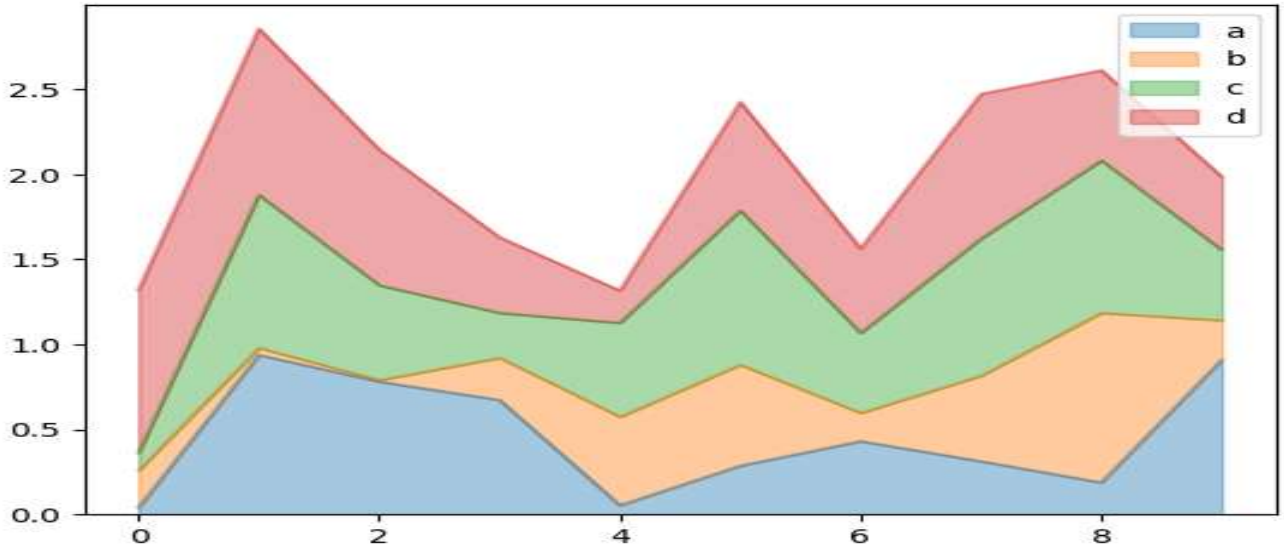
Explanation: plot() method by default creates a line plot for all numeric columns in the DataFrame, using the index for the x-axis.

2. Area Plots using Pandas DataFrame

Area plot shows data with a line and fills the space below the line with color. It helps see how things change over time. we can plot it using **DataFrame.plot.area() f**unction.

```
df2.plot.area(alpha=0.4)
```

Output:



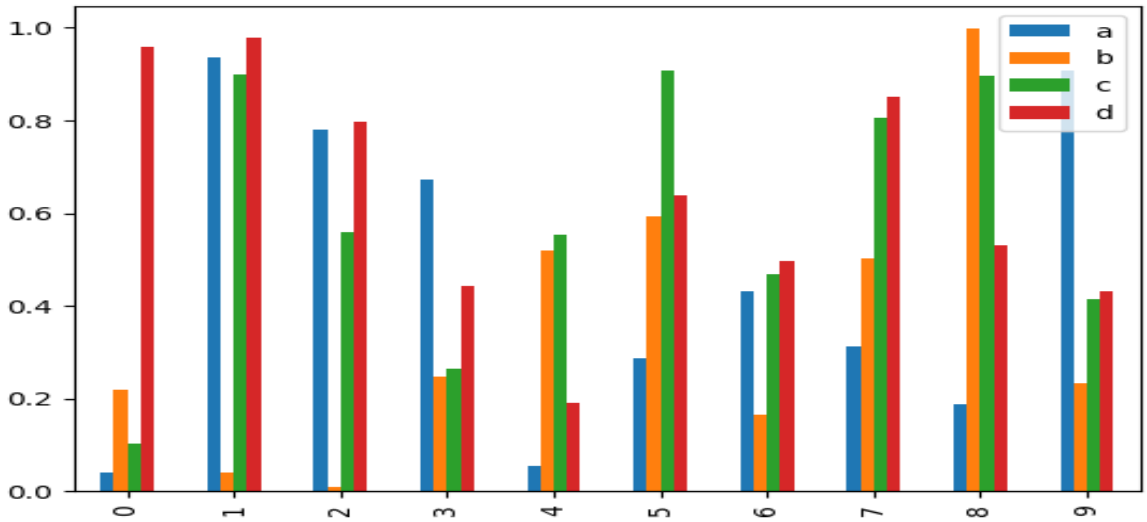
Explanation: **plot.area()** creates an area chart by filling space under lines for each numeric column. **alpha=0.4** sets transparency to make overlaps clearer.

3. Bar Plots using Pandas DataFrame

A bar chart presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally with **DataFrame.plot.bar()** function.

```
df2.plot.bar()
```

Output:



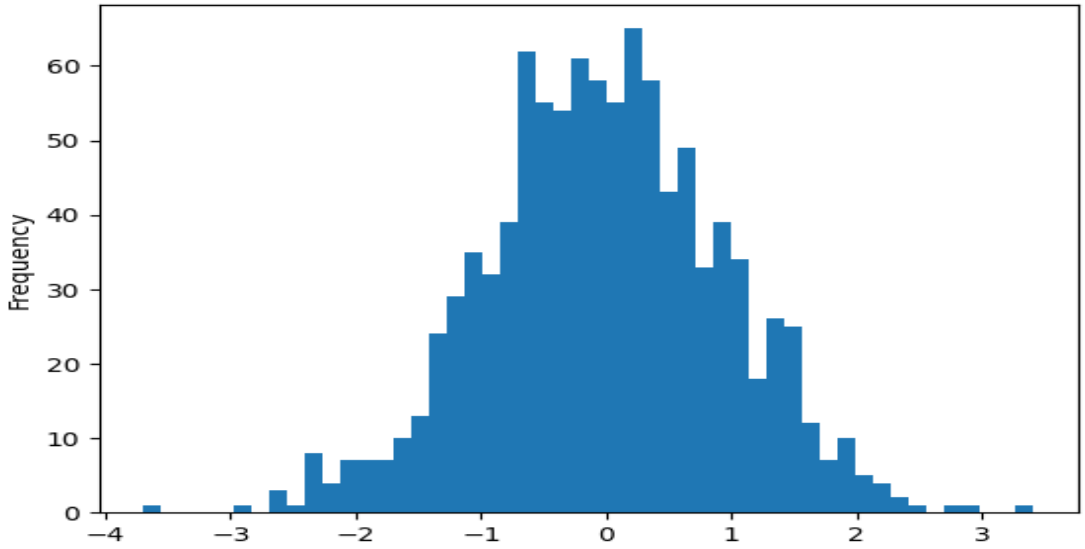
Explanation: plot.bar() creates a vertical bar chart showing values for each category or index.

4. Histogram Plot using Pandas DataFrame

Histograms help visualize the distribution of data by grouping values into bins. Pandas use **DataFrame.plot.hist()** function to plot histogram.

```
df1['A'].plot.hist(bins=50)
```

Output:

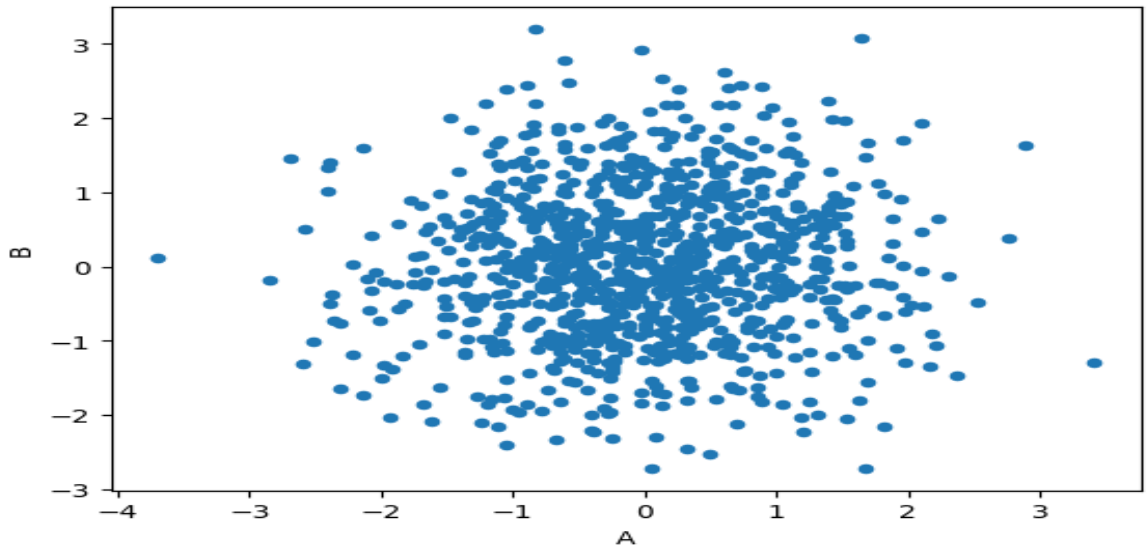


5. Scatter Plot using Pandas DataFrame

Scatter plots are used when you want to show the relationship between two variables. They are also called correlation and can be created using **DataFrame.plot.scatter()** function.

```
df1.plot.scatter(x ='A', y ='B')
```

Output:



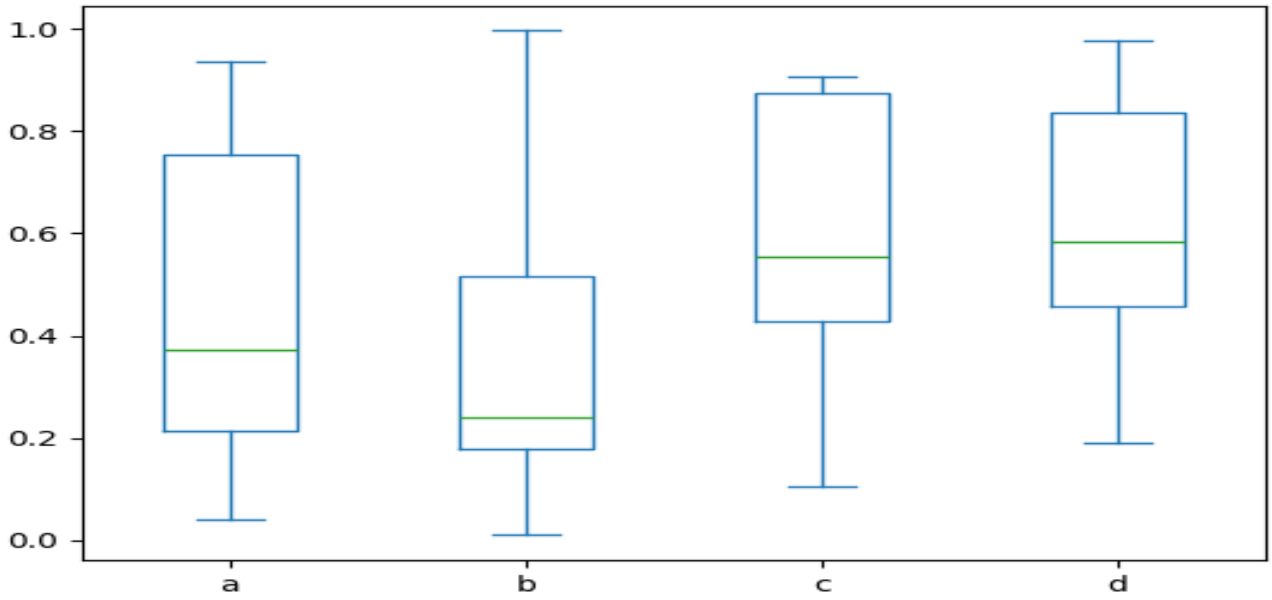
Explanation: **plot.scatter()** creates a scatter plot to show the relationship between two numeric columns. **x** and **y** specify the columns for the x-axis and y-axis.

6. Box Plots using Pandas DataFrame

A box plot displays the distribution of data, showing the median, quartiles, and outliers. we can use **DataFrame.plot.box()** function or **DataFrame.boxplot()** to create it.

```
df2.plot.box()
```

Output:



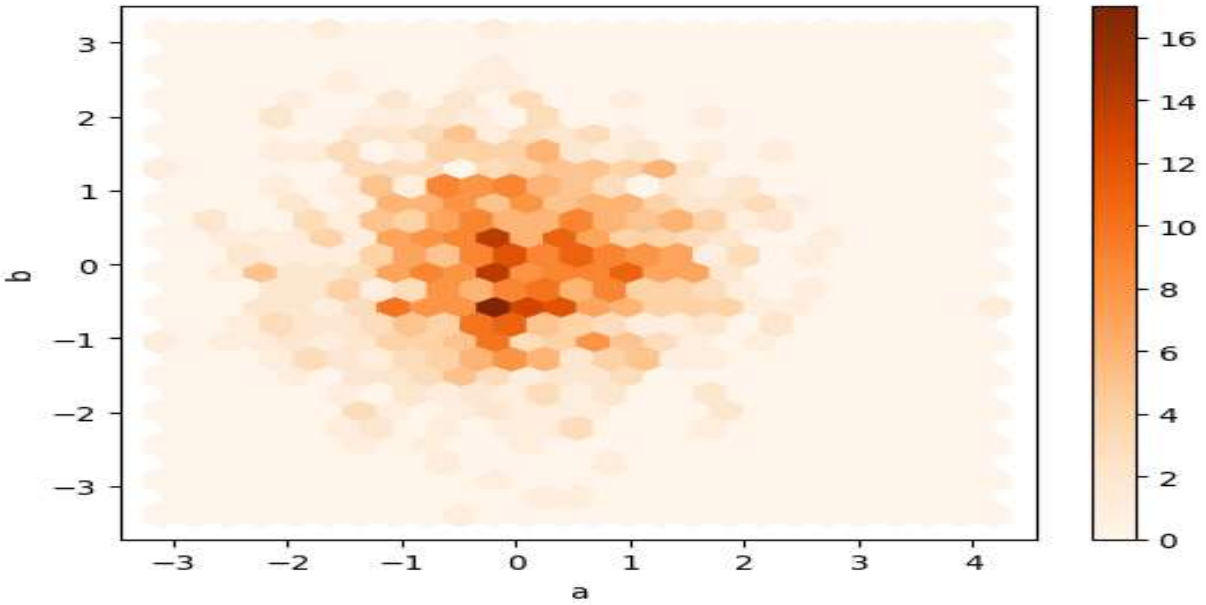
Explanation: plot.box() generates a box-and-whisker plot, visualizing median, quartiles and outliers.

7. Hexagonal Bin Plots using Pandas DataFrame

Hexagonal binning helps manage dense datasets by using hexagons instead of individual points. It's useful for visualizing large datasets where points may overlap. Let's create the hexagonal bin plot.

```
df.plot.hexbin(x ='a', y ='b', gridsize = 25, cmap ='Oranges')
```

Output:



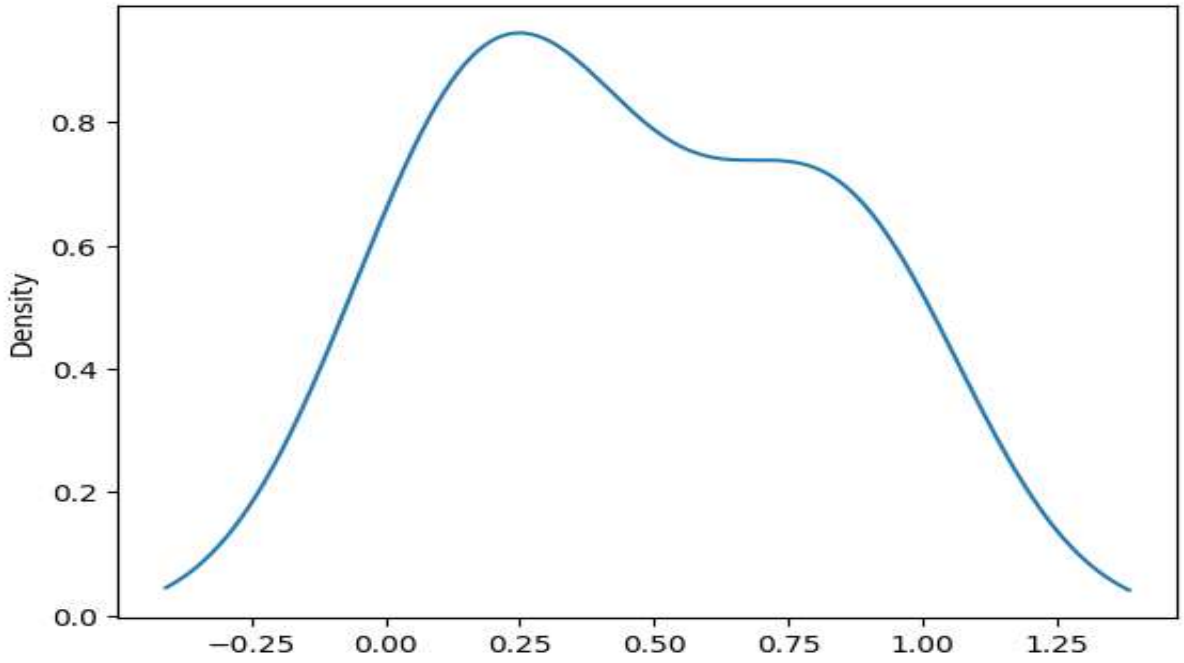
Explanation: plot.hexbin() creates a hexagonal bin plot for dense scatter data. **x** and **y** set the axes, gridsize controls hexagon count and cmap defines the color based on density.

8. Kernel Density Estimation plot (KDE) using Pandas DataFrame

KDE (Kernel Density Estimation) creates a smooth curve to show the shape of data by using the **df.plot.kde()** function. It's useful for visualizing data patterns and simulating new data based on real examples.

```
df2['a'].plot.kde()
```

Output:



Explanation: plot.kde() creates a Kernel Density Estimation plot, showing a smooth probability density curve.

Customizing Plots

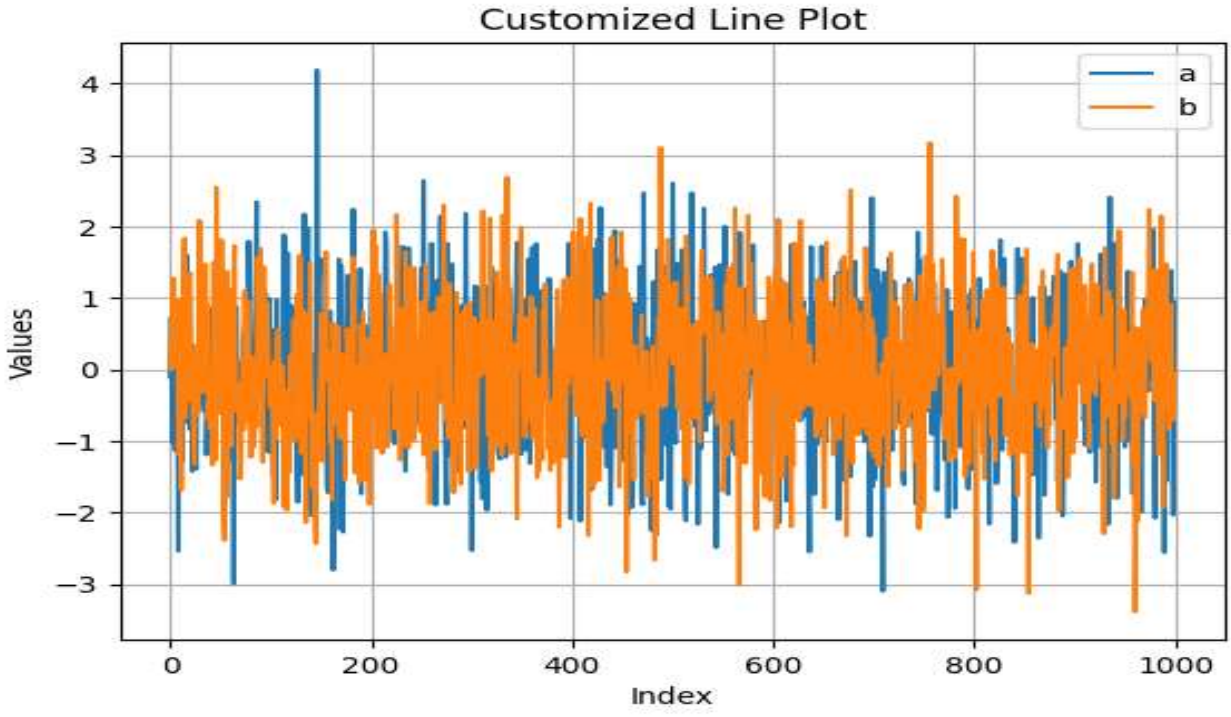
Pandas allows you to customize your plots in many ways. You can change things like colors, titles, labels, and more. Here are some common customizations.

1. Adding a Title, Axis Labels and Gridlines

You can customize the plot by adding a title and labels for the x and y axes. You can also enable gridlines to make the plot easier to read:

```
df.plot(title='Customized Line Plot', xlabel='Index', ylabel='Values', grid=True)
```

Output:

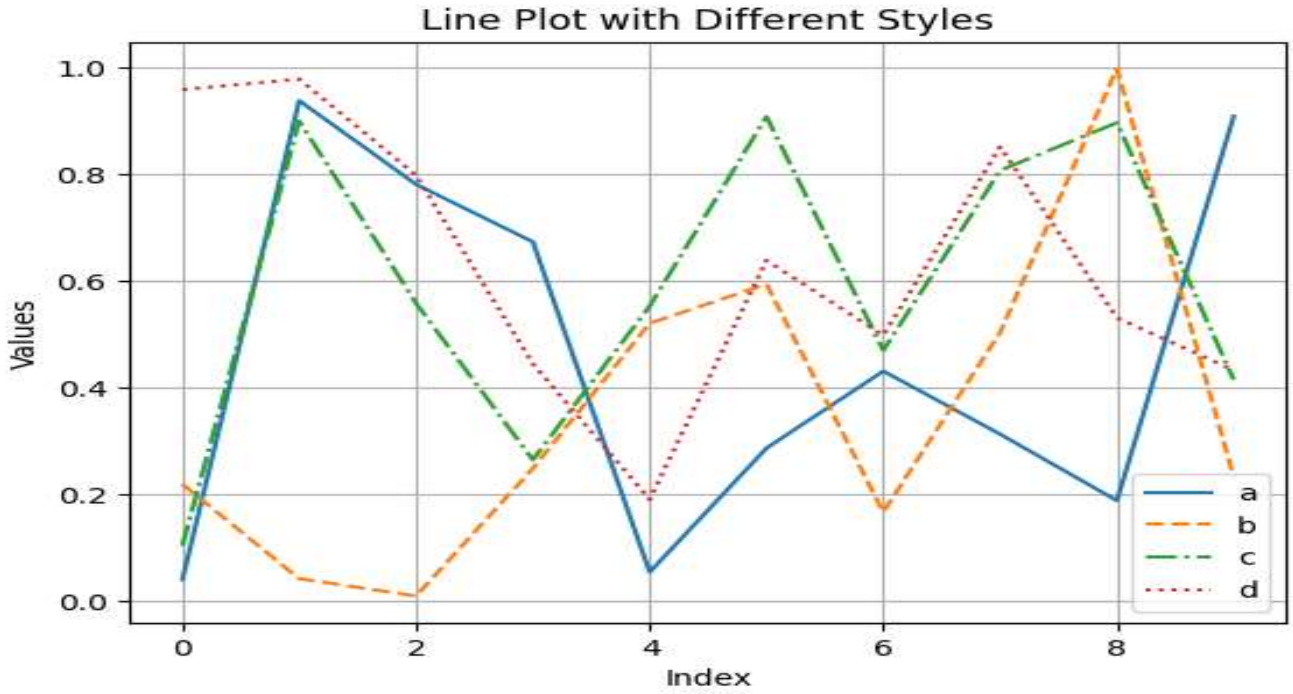


2. Line Plot with Different Line Styles

If you want to differentiate between the two lines visually you can change the line style (e.g., solid line, dashed line) with the help of pandas**.**

```
df.plot(style=['-', '--', '-.', ':'], title='Line Plot with Different Styles', xlabel='Index', ylabel='Values', grid=True)
```

Output:

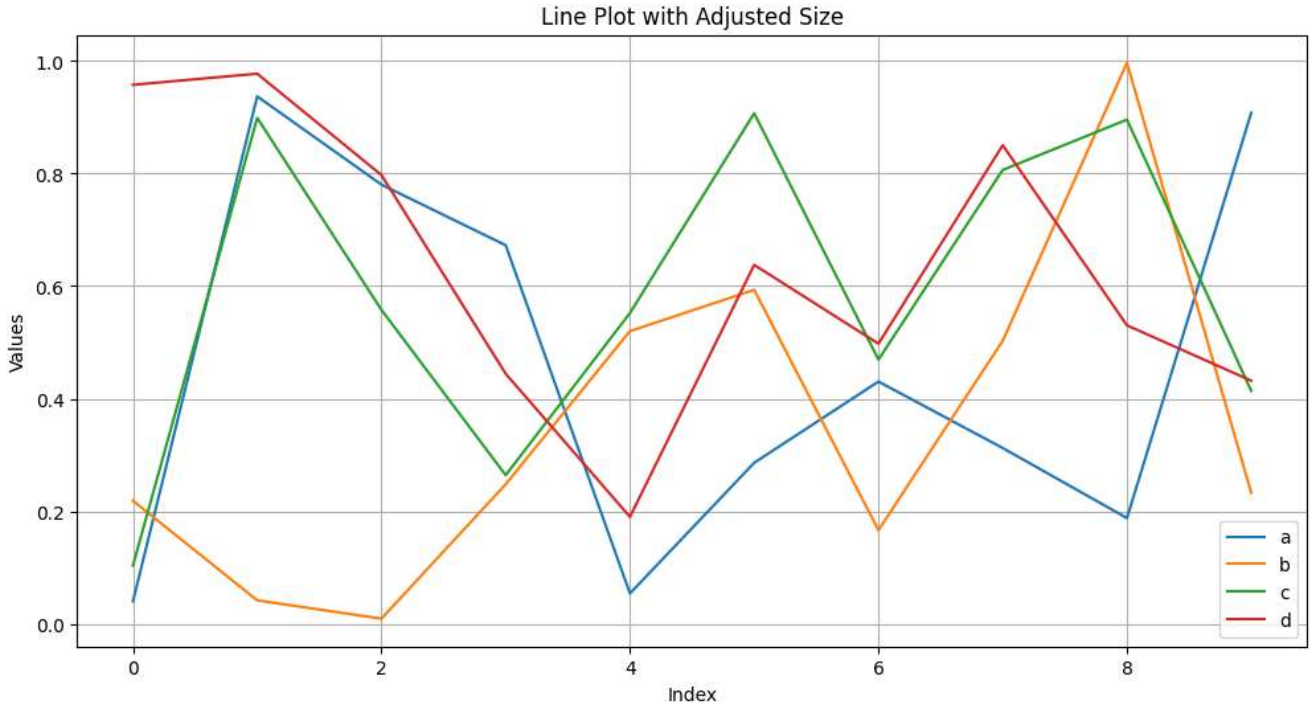


3. Adjusting the Plot Size

Change the size of the plot to better fit the presentation or analysis context You can change it by using the **figsize** parameter:

```
df.plot(figsize=(12, 6), title='Line Plot with Adjusted Size', xlabel='Index', ylabel='Values', grid=True)
```

Output:

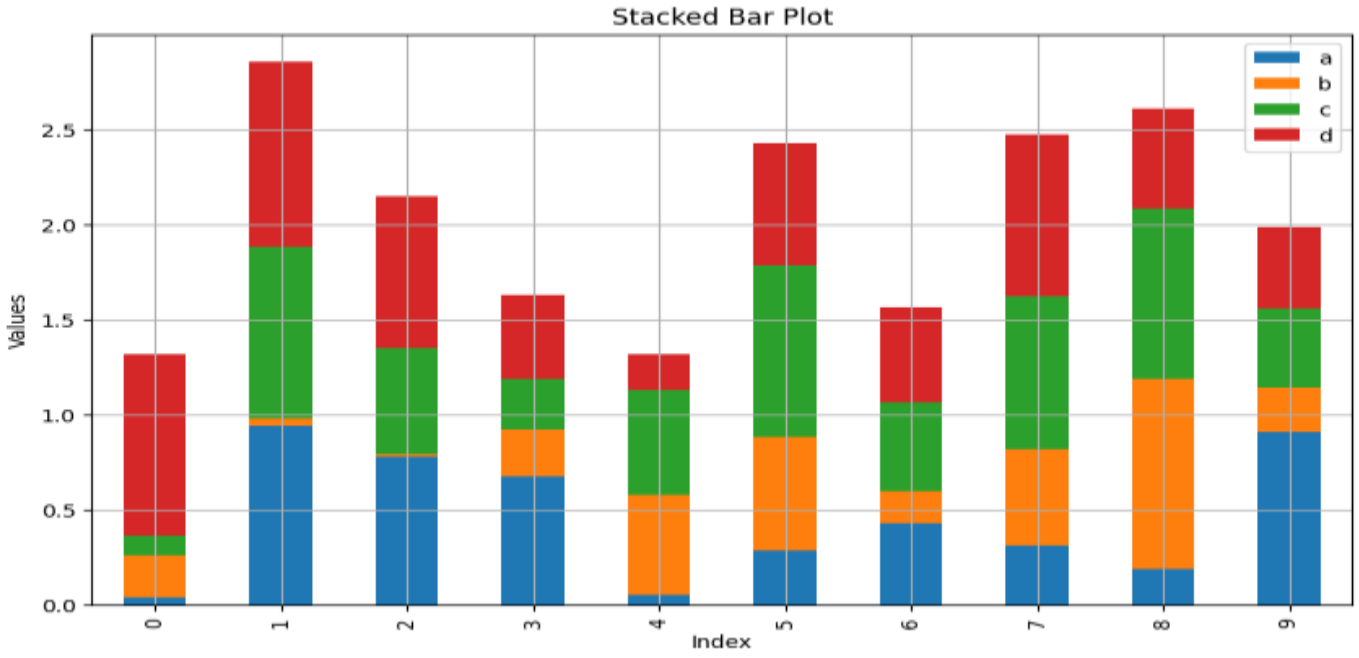


4. Stacked Bar Plot

A stacked bar plot can be created by setting **stacked=True**. It helps you visualize the cumulative value for each index.

```
df.plot.bar(stacked=True, figsize=(10, 6), title='Stacked Bar Plot', xlabel='Index', ylabel='Values', grid=True)
```

Output:



Pandas DataFrame corr() Method

Pandas **dataframe.corr()** is used to find the pairwise correlation of all columns in the Pandas Dataframe in Python. Any NaN values are automatically excluded. To ignore any non-numeric values, use the parameter numeric_only = True. In this article, we will learn about DataFrame.corr() method in Python.

Pandas DataFrame corr() Method Syntax

Syntax: DataFrame.corr(self, method='pearson', min_periods=1, numeric_only = False)

Parameters:

- **method:**
  - pearson: standard correlation coefficient
  - kendall: Kendall Tau correlation coefficient
  - spearman: Spearman rank correlation
- **min_periods:** Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation
- **numeric_only**: Whether only the numeric values are to be operated upon or not. It is set to False by default.

Pandas Data Correlations corr() Method

A good correlation depends on the use, but it is safe to say you have at least 0.6 (or -0.6) to call it a good correlation. A simple example to show how correlation work in Python.

```
import pandas as pd

df = {
  "Array_1": [30, 70, 100],
  "Array_2": [65.1, 49.50, 30.7]
}

data = pd.DataFrame(df)

print(data.corr())
```

Output

```
          Array_1                                          Array_2
Array_1                                  1.000000          -0.990773
Array_2 -0.990773  1.000000
```

Creating Sample Dataframe

```
# importing pandas as pd
import pandas as pd

# Making data frame from the csv file
df = pd.read_csv("nba.csv")

# Printing the first 10 rows of the data frame for visualization
df[:10]
```

Output

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
5	Amir Johnson	Boston Celtics	90.0	PF	29.0	6-9	240.0	NaN	12000000.0
6	Jordan Mickey	Boston Celtics	55.0	PF	21.0	6-8	235.0	LSU	1170960.0
7	Kelly Olynyk	Boston Celtics	41.0	C	25.0	7-0	238.0	Gonzaga	2165160.0
8	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville	1824360.0
9	Marcus Smart	Boston Celtics	36.0	PG	22.0	6-4	220.0	Oklahoma State	3431040.0

Python Pandas DataFrame corr() Method Examples

Find Correlation Among the Columns Using pearson Method

Here, we are using corr() function to find the correlation among the columns in the Dataframe using 'Pearson' method. We are only having four numeric columns in the Dataframe. The output Dataframe can be interpreted as for any cell, row variable correlation with the column variable is the value of the cell. As mentioned earlier, the correlation of a variable with itself is 1. For that reason, all the diagonal values are 1.00.

```
# To find the correlation among
# the columns using pearson method
df.corr(method='pearson')
```

Output

	Number	Age	Weight	Salary
Number	1.000000	0.028724	0.206921	-0.112386
Age	0.028724	1.000000	0.087183	0.213459
Weight	0.206921	0.087183	1.000000	0.138321
Salary	-0.112386	0.213459	0.138321	1.000000

Find Correlation Among the Columns Using Kendall Method

Use Pandas df.corr() function to find the correlation among the columns in the Dataframe using 'kendall' method. The output Dataframe can be interpreted as for any cell, row variable correlation with the column

variable is the value of the cell. As mentioned earlier, the correlation of a variable with itself is 1. For that reason, all the diagonal values are 1.00.

```
# importing pandas as pd
import pandas as pd

# Making data frame from the csv file
df = pd.read_csv("nba.csv")

# To find the correlation among
# the columns using kendall method
df.corr(method='kendall')
```

Output

	Number	Age	Weight	Salary
Number	1.000000	0.005536	0.155850	-0.075301
Age	0.005536	1.000000	0.066130	0.172616
Weight	0.155850	0.066130	1.000000	0.087165
Salary	-0.075301	0.172616	0.087165	1.000000