# 21CSS303T
# DATA SCIENCE
# UNIT-1
# INTRODUCTION TO DATA SCIENCE, NUMPY AND PANDAS
# PART-2

- Pandas
- Exploring Data using Series
- Exploring Data using DataFrames
- Index objects
- Re index
- Drop Entry
- Selecting Entries
- Data Alignment
- Rank and Sort
- Summary Statistics
- Index Hierarchy
- Data Acquisition: Gather information from different sources
- Web APIs
- Open Data Sources
- Web Scrapping

## INTRODUCTION TO PANDAS

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "**Panel Data**", and "**Python Data Analysis**" and was created by Wes **McKinney in 2008**.
- Pandas allow us to analyze big data and make conclusions based on statistical theories.

```
pip install pandas
import pandas
import pandas as pd
```

## DATA STRUCTURE IN PANDAS

1. Series
2. Dataframe

## EXPLORING DATA USING SERIES

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.

**Syntax: pandas.Series(data=None, index=None, dtype=None, name=None, copy=False)**
**Parameters:**

**data**: array- Contains data stored in Series.
**index**: array-like or Index (1d)
**dtype**: str, numpy.dtype, or ExtensionDtype, optional
**name:** str, optional
**copy:** bool, default False

**Example: Create a simple Pandas Series from a list:**

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)
```

**Output:**

```
0    1
1    7
2    2
```

## Labels

```
print(myvar[0])
```

**Output:**

```
1
```

## Create Labels

With the index argument, you can name your own labels.

**Example: Create your own labels**

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
```

**Output:**

```
x    1
y    7
z    2
```

***When you have created labels, you can access an item by referring to the label.***

**Example:**

```
print(myvar["y"])
```

**Output:** 7

## Key/Value Objects as Series

You can also use a key/value object, like a dictionary, when creating a Series.

**Example: Create a simple Pandas Series from a dictionary:**

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

**Output:**

```
day1    420
day2    380
day3    390
```

***Note: The keys of the dictionary become the labels.***

To select only some of the items in the dictionary, use the index argument and specify only the items you want to include in the Series.

**Example: Create a Series using only data from "day1" and "day2":**

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
```

```
myvar = pd.Series(calories, index = ["day1", "day2"])
print(myvar)
```

**Output:**

```
day1    420
day2    380
```

### DataFrames from Series

- Data sets in Pandas are usually multi-dimensional tables, called DataFrames.
- Series is like a column, a DataFrame is the whole table.

**Example: Create a DataFrame from two Series**

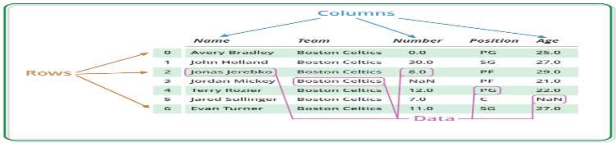
```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
myvar = pd.DataFrame(data)
print(myvar)
```

**Output:**

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

# EXPLORING DATA USING DATAFRAMES

- Pandas DataFrame is a **two-dimensional size-mutable**, potentially **heterogeneous tabular data structure with labeled axes (rows and columns).**
- A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns like a spreadsheet or SQL table, or a dict of Series objects.
- Pandas DataFrame consists of three principal components: **data, rows, and columns**.



| | Name | Team | Number | Position | Age |
|---|---|---|---|---|---|
| 0 | Avery Bradley | Boston Celtics | 0.0 | PG | 25.0 |
| 1 | John Holland | Boston Celtics | 30.0 | SG | 27.0 |
| 2 | Jonas Jerebko | Boston Celtics | 8.0 | PF | 29.0 |
| 3 | Jordan Mickey | Boston Celtics | NaN | PF | 21.0 |
| 4 | Terry Rozier | Boston Celtics | 12.0 | PG | 22.0 |
| 5 | Jared Sullinger | Boston Celtics | 7.0 | C | NaN |
| 6 | Evan Turner | Boston Celtics | 11.0 | SG | 27.0 |

**Example:**

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

**Output:**

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

# SELECTING ENTRIES

### a) Locate Row

**Example: Return row 0:**

```
print(df.loc[0])
```

**Output**:

```
 calories    420
 duration     50
```

**Example: Return row 0 and 1:**

```
print(df.loc[[0, 1]])
```

**Output:**

```
       calories  duration
 0      420       50
 1      380       40
```

***Note: When using [], the result is a Pandas DataFrame.***

### b) Named Indexes

```
import pandas as pd
data = {   "calories": [420, 380, 390],   "duration": [50, 40, 45] }
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

**Output:**

```
           calories  duration
 day1       420        50
 day2       380        40
 day3       390        45
```

### Locate Named Indexes

**Example: Return "day2":**

```
print(df.loc["day2"])
```

**Output:**

```
 calories    380
 duration     40
```

## ACCESSING AND SLICING OF SERIES AND DATAFRAMES (SELECTING ENTRIES)

**1. Accessing** is used to select a specific row or column from a DataFrame.

- **Using Integer Indexing**

**Example:**

```
import pandas as pd
data = pd.Series([10, 20, 38, 40, 50])
print(data[2])
```

**Output:** 38

- **Using Custom Index Label**

```
import pandas as pd
data = pd.Series([10, 20, 28, 40, 50], index=['A', 'B', 'C', 'D', 'E'])
print(data['C'])
```

**Output:** 28

- **Using Multiple Column Names**

```
import pandas as pd
```

```
data = {'Name': ['John', 'Anna', 'Peter'], 'Age': [28, 24, 22]}
df = pd.DataFrame(data)
```

**Output:**

```
   Name  Age
0   John   28
1   Anna   24
2  Peter   22
```

2. **Slicing:** allowing extraction of specific data subsets based on integer positions.

**Example:**

```
import pandas as pd
data = pd.DataFrame({'Brand': ['Maruti', 'Hyundai', 'Tata','Mahindra', 'Maruti', 'Hyundai', 'Renault',
'Tata', 'Maruti'],  'Year': [2012, 2014, 2011, 2015, 2012, 2016, 2014, 2018, 2019],
'Kms Driven': [50000, 30000, 60000, 25000, 10000, 46000, 31000, 15000, 12000],
'City': ['Gurgaon', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai', 'Chennai', 'Ghaziabad'],
'Mileage': [28, 27, 25, 26, 28, 29, 24, 21, 24]})
print(data)
```

- **loc()**

The loc() function is label based data selecting method which means that we have to pass the name of the row or column which we want to select.

**Example: Selecting Data According to Some Conditions**

```
print(data.loc[(data.Brand == 'Maruti') & (data.Mileage > 25)])
```

**Output:**

```
   Brand  Year  Kms Driven      City  Mileage
0  Maruti  2012      50000   Gurgaon      28
4  Maruti  2012      10000    Mumbai      28
```

**Example: Selecting a Range of Rows from the DataFrame**

```
print(data.loc[2: 5])
```

**Output:**

```
       Brand  Year  Kms Driven    City  Mileage
2     Tata  2011      60000  Mumbai      25
3  Mahindra  2015      25000   Delhi      26
4   Maruti  2012      10000  Mumbai      28
5  Hyundai  2016      46000   Delhi      29
```

**Example: Updating the Value of Any Column**

```
data.loc[(data.Year < 2015), ['Mileage']] = 22
print(data)
```

**Output:**

```
    Brand  Year  Kms Driven      City  Mileage
0   Maruti  2012      50000   Gurgaon      22
1  Hyundai  2014      30000     Delhi      22
2     Tata  2011      60000    Mumbai      22
3  Mahindra  2015      25000     Delhi      26
4   Maruti  2012      10000    Mumbai      22
5  Hyundai  2016      46000     Delhi      29
6  Renault  2014      31000    Mumbai      22
7     Tata  2018      15000   Chennai      21
```

```
8   Maruti  2019       12000  Ghaziabad       24
```

- ## iloc()

The iloc() function is an indexed-based selecting method which means that we have to pass an integer index in the method to select a specific row/column.

**Example: Selecting Rows Using Integer Indices**

```
print(data.iloc[[0, 2, 4, 7]])
```

**Output:**

```
    Brand  Year  Kms Driven       City  Mileage
0   Maruti  2012       50000    Gurgaon       28
2     Tata  2011      60000     Mumbai       25
4   Maruti  2012       10000     Mumbai       28
7     Tata  2018      15000    Chennai       21
```

**Example: Selecting a Range of Columns and Rows Simultaneously**

```
print(data.iloc[1: 5, 2: 5])
```

**Output:**

```
    Kms Driven    City  Mileage
1      30000   Delhi      27
2      60000  Mumbai       25
3      25000   Delhi      26
4      10000  Mumbai       28
```

- ## at[]

Pandas at[] is used to return data in a dataframe at the passed location.

**Syntax: Dataframe.at[position, label]**

**Parameters:**

**position**: Position of element in column

**label**: Column name to be used

**Return type**: Single element at passed position

**Example:**

```
position = 2
label = 'Brand'
output = data.at[position, label]
print(output)
```

**Output:** Tata

.at and .iat are like .loc and .iloc but they are faster for single value retreval where as others are faster for selection and slicing

- ## iat[]

Pandas iat[] method is used to return data in a dataframe at the passed location.

**Syntax: Dataframe.iat[row, column]**

**Parameters:**

**position**: Position of element in column

**label**: Position of element in row

**Return type:** Single element at passed position

**Example:**

```
column = 3
row = 2
output = data.iat[row, column]
print(output)
```

**Output:** Mumbai

# INDEXING

Indexing in pandas means simply **selecting particular rows and columns** of data from a DataFrame. Indexing could mean **selecting all the rows and some of the columns**, some of the rows and all of the columns, or some of each of the rows and columns. Indexing can also be known as **Subset Selection**.

1. **Dataframe.[ ]** : This function also known as indexing operator
2. **Dataframe.loc[ ] :** This function is used for labels.
3. **Dataframe.iloc[ ] :** This function is used for positions or integer based
4. **Dataframe.ix[] :** This function is used for both label and integer based

*Collectively, they are called the **indexers**.*

### 1. Indexing a Dataframe using indexing operator []:

- **Selecting a single column**

```
import pandas as pd
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
single_col_df = df[['A']]
print(single_col_df)
print(type(single_col_df)) # Output: <class 'pandas.core.frame.DataFrame'>
```

- **Selecting multiple columns**

```
import pandas as pd
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9], 'D': [10, 11, 12]}
df = pd.DataFrame(data)
selected_df = df[['A', 'C']]
print(selected_df)
```

### 2. Indexing a DataFrame using .loc[ ]

- **Selecting a single row**

```
import pandas as pd
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data)
single_row_df = df.loc[[1]] -> Dataframe        df.loc[1] -> Tupple
print(single_row_df)
```

- **Selecting multiple rows**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12]}
df = pd.DataFrame(data)
multiple_rows_df = df.loc[[1, 3]]
print(multiple_rows_df)
```

- **Selecting two rows and three columns**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12], 'D': [13, 14, 15, 16]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3', 'row4']) # Custom row labels
subset_df = df.loc[['row1', 'row3'], ['A', 'B', 'C']]
print(subset_df)
```

- **Selecting all of the rows and some columns**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12], 'D': [13, 14, 15, 16]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3', 'row4']) # Custom row labels
```

```
subset_df = df.loc[:, ['A', 'B', 'C']]
print(subset_df)
```

3. **Indexing a DataFrame using .iloc[ ] :**

- **Selecting a single row**

```
import pandas as pd
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])
single_row_df = df.iloc[[1]]
```

- **Selecting multiple rows**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3', 'row4']) # Custom row labels
multiple_rows_df = df.iloc[[0, 2]] # Selecting rows at index 0 and 2
```

- **Selecting two rows and two columns**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12], 'D': [13, 14, 15, 16]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3', 'row4']) # Custom row labels
subset_df = df.iloc[[0, 2], [0, 2]]
print(subset_df)
```

- **Selecting all the rows and some columns**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12], 'D': [13, 14, 15, 16]}
df = pd.DataFrame(data)
subset_df = df.iloc[:, [0, 2]] # Selecting all rows and columns at index 0 and 2
print(subset_df)
```

4. **Indexing a using Dataframe.ix[ ] :**

- **Selecting a single row using .ix[] as .loc[]**

```
import pandas as pd
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12], 'D': [13, 14, 15, 16]}
df = pd.DataFrame(data)
first = df.ix[‘A’]
print(first)
```

# REINDEX

- The main task of the Pandas reindex is to conform DataFrame to a new index with optional filling logic and to place NA/NaN in that location where the values are not present in the previous index.
- It returns a new object unless the new index is produced as an equivalent to the current one, and the value of copy becomes False.
- Reindexing is used to change the index of the rows and columns of the DataFrame. We can reindex the single or multiple rows by using the reindex() method. Default values in the new index are assigned NaN if it is not present in the DataFrame.

Syntax: **DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill_value=nan, limit=None, tolerance=None)**

Parameters:

- **labels**: It is an optional parameter that refers to the new labels or the index to conform to the axis that is specified by the 'axis'.
- **index, columns**: It is also an optional parameter that refers to the new labels or the index. It generally prefers an index object for avoiding the duplicate data.
- **axis**: It is also an optional parameter that targets the axis and can be either the axis name or the numbers.
- **method**: It is also an optional parameter that is to be used for filling the holes in the reindexed DataFrame. It can only be applied to the DataFrame or Series with a monotonically increasing/decreasing order.
    - **None**: It is a default value that does not fill the gaps.
    - **pad / ffill**: It is used to propagate the last valid observation forward to the next valid observation.
    - **backfill / bfill**: To fill the gap, It uses the next valid observation.
    - **nearest:** To fill the gap, it uses the next valid observation.
- **copy:** Its default value is True and returns a new object as a boolean value, even if the passed indexes are the same.
- **level**: It is used to broadcast across the level, and match index values on the passed MultiIndex level.
- **fill_value**: Its default value is np.NaN and used to fill existing missing (NaN) values. It needs any new element for successful DataFrame alignment, with this value before computation.
- **limit**: It defines the maximum number of consecutive elements that are to be forward or backward fill.
- **tolerance**: It is also an optional parameter that determines the maximum distance between original and new labels for inexact matches. At the matching locations, the values of the index should most satisfy the equation abs(index[indexer] ? target) <= tolerance.

Returns:

It returns reindexed DataFrame.

**Example**:

```
import pandas as pd
info1 =pd.DataFrame({"A":[1, 5, 3, 4, 2],
              "B":[3, 2, 4, 3, 4],
              "C":[2, 2, 7, 3, 4],
              "D":[4, 3, 6, 12, 7]})
info1.reindex(columns =["A", "B", "D", "E"])
```

Output:

```
      A    B    D    E
0    1    3    4    NaN
1    5    2    3    NaN
2    3    4    6    NaN
3    4    3    12   NaN
4    2    4    7    NaN
```

Notice that NaN values are present in the new columns after reindexing, we can use the argument fill_value to the function for removing the NaN values.

```
# fill the missing values by 37
info1.reindex(columns =["A", "B", "D", "E"], fill_value =37)
```

Output:

```
     A     B     D     E
0    1     3     4     37
1    5     2     3     37
2    3     4     6     37
3    4     3     12    37
4    2     4     7     37
```

# DROP ENTRY

- The drop() method removes the specified row or column.
- By specifying the column axis (axis='columns'), the drop() method removes the specified column.
- By specifying the row axis (axis='index'), the drop() method removes the specified row.

**Syntax:** dataframe.drop(labels, axis, index, columns, level, inplace., errors)

| Parameter | Value | Description |
|---|---|---|
| Labels | | Optional, The labels or indexes to drop. If more than one, specify them in a list |
| axis | 0<br>1<br>'index'<br>'columns' | Optional, Which axis to check, default 0. |
| index | String<br>Listz` | Optional, Specifies the name of the rows to drop. Can be used instead of the labels parameter. |
| columns | String<br>List | Optional, Specifies the name of the columns to drop. Can be used instead of the labels parameter. |
| level | Number<br>level name | Optional, default None. Specifies which level ( in a hierarchical multi index) to check along |
| inplace | TRUE<br>FALSE | Optional, default False. If True: the removing is done on the current DataFrame. If False: returns a copy where the removing is done. |
| errors | 'ignore'<br>'raise' | Optional, default 'ignore'. Specifies whether to ignore errors or not |

**Example:**

```
import pandas as pd
details = {'Name': ['Ankit', 'Aishwarya', 'Shaurya', 'Shivangi'],'Age': [23, 21, 22, 21],'University': ['BHU', 'JNU', 'DU', 'BHU']}
df = pd.DataFrame(details, columns=['Name', 'Age', 'University'],index=['a', 'b', 'c', 'd'])
print(df)
```

**Output**:

```
      Name  Age University
a     Ankit   23       BHU
b  Aishwarya   21       JNU
c    Shaurya   22        DU
d   Shivangi   21       BHU
```

### 1. Using drop() Method:

**A. Delete a Single Row in DataFrame by Row Index Label**

```
update_df = df.drop('c')
print(update_df)
```

**Output:**

```
        Name            Age   University
a       Ankit           23      BHU
b       Aishwarya       21      JNU
d       Shivangi        21      BHU
```

B. **Delete Multiple Rows in DataFrame by Index Labels**

```
update_df = df.drop(['b', 'c'])
print(update_df)
```

**Output:**

```
   Name          Age     University
a  Ankit       23          BHU
d  Shivangi   21           BHU
```

### 2. drop method with index parameter

A. **drop with index**

```
# return a new dataframe by dropping a row 'b' & 'c' from dataframe using their respective index
position
update_df = df.drop([df.index[1], df.index[2]])
print(update_df)
```

**Output:**

```
     Name     Age  University
a      Ankit   23         BHU
d  Shivangi  21          BHU
```

B. **Dropping Rows with inplace=True**

```
# dropping a row 'c' & 'd' from actual dataframe
df.drop(['c', 'd'], inplace=True)
print(df)
```

**Output:**

```
        Name            Age     University
a        Ankit        23        BHU
b       Aishwarya       21      JNU
```

### 3. Drop column

```
#Remove the "age" column from the DataFrame
updated_df = df.drop("Age", axis='columns')
print(updated_df)
```

**Output:**

```
        Name            University
a       Ankit           BHU
b       Aishwarya       JNU
c       Shaurya         DU
d       Shivangi        BHU
```

# DATA ALIGNMENT

In pandas, data alignment refers to the automatic alignment of data when performing operations on objects like Series and DataFrame. This ensures that calculations are performed correctly even when indices do not match.

### 1. Data Alignment in Series

When performing operations on Series objects, pandas aligns them based on their index labels.

**Example**:

```
import pandas as pd
s1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
s2 = pd.Series([5, 15, 25], index=['b', 'c', 'd'])
```

```
        result = s1 + s2
        print(result)
```

**Output:**

```
a NaN # 'a' is only in s1
b 25.0 # 20 + 5
c 45.0 # 30 + 15
d NaN # 'd' is only in s2
dtype: float64
```

### 2. Data Alignment in DataFrame

For DataFrame, data alignment occurs along both rows (index) and columns.

**Example**:

```
        df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]}, index=['x', 'y'])
        df2 = pd.DataFrame({'B': [5, 6], 'C': [7, 8]}, index=['y', 'z'])
        result = df1 + df2
        print(result)
```

**Output:**

```
A B C
x NaN NaN NaN # 'x' is only in df1
y NaN 9.0 NaN # 'y' is in both
z NaN NaN NaN # 'z' is only in df2
```

### 3. Align column to Left

```
        left_aligned_df = df.style.set_properties(**{'text-align': 'left'})
        display(left_aligned_df)
```

# RANK

- Pandas Dataframe.rank() method returns a rank of every respective index of a series passed. The rank is returned on the basis of position after sorting.

Syntax: **DataFrame.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)**

**Parameters**:

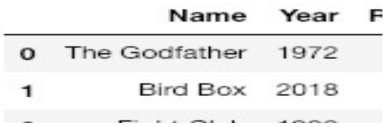
- **axis:** 0 or 'index' for rows and 1 or 'columns' for Column.
- **method:** Takes a string input('average', 'min', 'max', 'first', 'dense') which tells pandas what to do with same values. Default is average which means assign average of ranks to the similar values.
- **numeric_only:** Takes a boolean value and the rank function works on non-numeric value only if it's False.
- **na_option**: Takes 3 string input('keep', 'top', 'bottom') to set position of Null values if any in the passed Series.
- **ascending**: Boolean value which ranks in ascending order if True.
- **pct**: Boolean value which ranks percentage wise if True.

Return type: Series with Rank of every index of caller series.

**Example:**

```
        import pandas as pd
        movies = {'Name': ['The Godfather', 'Bird Box', 'Fight Club'],
                        'Year': ['1972', '2018', '1999'],
                        'Rating': ['9.2', '6.8', '8.8']}
        df = pd.DataFrame(movies)
        print(df)
```

Output:

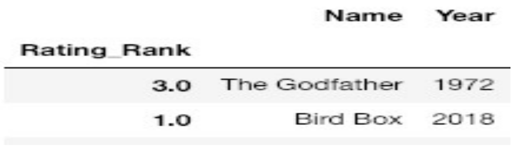


|  | Name | Year | F |
|---|---|---|---|
| 0 | The Godfather | 1972 |  |
| 1 | Bird Box | 2018 |  |

**Example:**

```
df['Rating_Rank'] = df['Rating'].rank(ascending = 1)
df = df.set_index('Rating_Rank')
print(df)
```

Output:

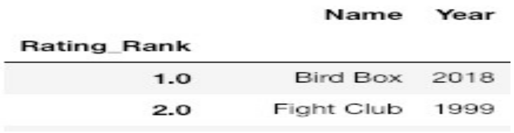


|  | Name | Year |
|---|---|---|
| Rating_Rank |  |  |
| 3.0 | The Godfather | 1972 |
| 1.0 | Bird Box | 2018 |

Example: **Sort the dataFrame based on the index**

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

Output:



|  | Name | Year |
|---|---|---|
| Rating_Rank |  |  |
| 1.0 | Bird Box | 2018 |
| 2.0 | Fight Club | 1999 |

# SORT

**pandas.DataFrame.sort_values**

**DataFrame.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)**

Parameters

- **by**: Single/List of column names to sort Data Frame by.
- **axis**: 0 or 'index' for rows and 1 or 'columns' for Column.
- **ascending**: Boolean value which sorts Data frame in ascending order if True.
- **inplace**: Boolean value. Makes the changes in passed data frame itself if True.
- **kind**: String which can have three inputs ('quicksort', 'mergesort' or 'heapsort') of algorithm used to sort data frame.
- **na_position**: Takes two string input 'last' or 'first' to set position of Null values. Default is 'last'.
- **ignore_indexbool**: default False. If True, the resulting axis will be labeled 0, 1, …, n - 1.
- **Keycallable**: optional. Apply the key function to the values before sorting. This is similar to the key argument in the builtin sorted() function, with the notable difference that this key function should be vectorized. It should expect a Series and return a Series with the same shape as the input. It will be applied to each column in by independently.

**Return Type**:
Returns a sorted Data Frame with Same dimensions as of the function caller Data Frame

### a. Sorting by Index

Use sort_index() to sort rows or columns by their index.

```
df = pd.DataFrame({'A': [3, 1, 2]}, index=['b', 'c', 'a'])
sorted_df = df.sort_index()
print(sorted_df)
```

Output:

```
# A
# a 2
# b 3
# c 1
```

### b. Sorting by Values

Use sort_values() to sort rows by column values.

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

Output:

```
# A
# c 1
# a 2
# b 3
```

### c. Sorting in Descending Order

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

## SUMMARY STATISTICS

Pandas provides several functions to compute descriptive statistics.

### 1. Aggregation Functions

These return a single value for each column or row.

- **Mean**: df.mean()
- **Median**: df.median()
- **Mode**: df.mode()
- **Standard deviation**: df.std()
- **Variance**: df.var()
- **Min/Max**: df.min(), df.max()
- **Sum**: df.sum()
- **Count**: df.count()
- **Quantiles**: df.quantile([0.25, 0.5, 0.75])

**Example:**

```
df = pd.DataFrame({
'A': [1, 2, 3],
'B': [4, 5, 6],
'C': [7, 8, 9]
})
print(df.mean()) # Column-wise mean
```

Output:

```
 A 2.0
 B 5.0
 C 8.0
```

```
print(df.mean(axis=1)) # Row-wise mean
```

### 2. Cumulative Functions

Cumulative functions return cumulative sums, products, or counts.

```
print(df['A'].cumsum())
```

Output:

```
0 1
1 3
2 6
```

### 3. Describe Method

The describe() method generates a summary of statistics for numerical columns.

```
print(df.describe())
```

Output:

```
  A B C
count 3.000000 3.0 3.0
mean 2.000000 5.0 8.0
std 1.000000 1.0 1.0
min 1.000000 4.0 7.0
25% 1.500000 4.5 7.5
50% 2.000000 5.0 8.0
75% 2.500000 5.5 8.5
max 3.000000 6.0 9.0
```

### 4. Value Counts

To count occurrences of unique values:

```
s = pd.Series([1, 2, 2, 3, 3, 3])
print(s.value_counts())
```

Output:

```
3 3
2 2
1 1
```

### 5. Correlation and Covariance

Correlation matrix: df.corr()

Covariance matrix: df.cov()

# INDEX HIERARCHY HIERARCHICAL INDEXING AND LEVELLING

- To make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index.
- Higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

**Syntax**:

***pandas.MultiIndex(levels=None, codes=None, sortorder=None, names=None, dtype=None, copy=False, name=None, verify_integrity=True)***

**Parameters:**

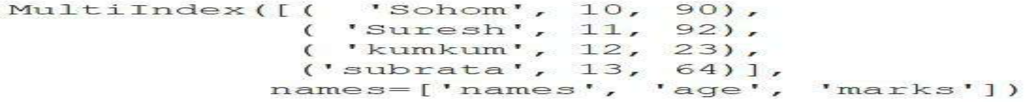
- **levels**: It is a sequence of arrays that shows the unique labels for each level.
- **codes**: It is also a sequence of arrays where integers at each level help us to designate the labels in that location.
- **sortorder**: optional int. It helps us to sort the levels lexicographically.
- **dtype**: data-type (size of the data which can be of 32 bits or 64 bits)
- **copy**: It is a boolean type parameter with a default value of False. It helps us to copy the metadata.
- **verify_integrity**: It is a boolean type parameter with a default value of True. It checks the integrity of the levels and codes i.t if they are valid.

### Creating a MultiIndex

### 1. Creating multi-index from arrays

```
import pandas as pd
arrays = ['Sohom','Suresh','kumkum','subrata']
age= [10, 11, 12, 13]
marks=[90,92,23,64]
multi_index = pd.MultiIndex.from_arrays([arrays,age,marks], names=('names', 'age','marks'))
print(multi_index)
```

**Output**:



```
MultiIndex([(   'Sohom', 10, 90),
            ( 'Suresh', 11, 92),
            ( 'kumkum', 12, 23),
            ('subrata', 13, 64)],
           names=['names', 'age', 'marks'])
```

**2. MultiIndex.from_tuples**

```
arrays = [
 ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
 ["one", "two", "one", "two", "one", "two", "one", "two"],
 ]
tuples = list(zip(*arrays))
print(tuples)
```

**Output:**

```
[('bar', 'one'),
('bar', 'two'),
('baz', 'one'),
('baz', 'two'),
('foo', 'one'),
('foo', 'two'),
('qux', 'one'),
('qux', 'two')]
```

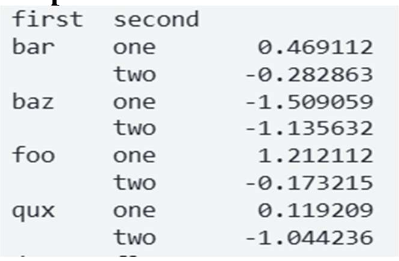
```
index = pd.MultiIndex.from_tuples(tuples, names=["first", "second"])
print(index)
```

**Output**:

```
MultiIndex([('bar', 'one'),
('bar', 'two'),
('baz', 'one'),
('baz', 'two'),
('foo', 'one'),
('foo', 'two'),
('qux', 'one'),
('qux', 'two')],
names=['first', 'second'])
```

```
s = pd.Series(np.random.randn(8), index=index)
print(s)
```

**Output**:

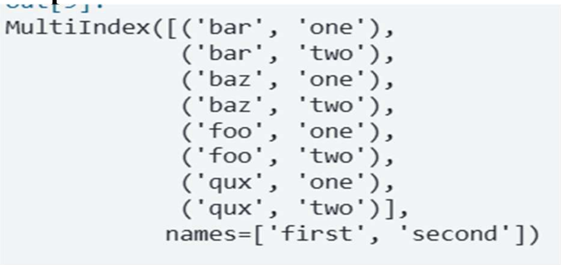


```
first  second
bar    one       0.469112
       two      -0.282863
baz    one      -1.509059
       two      -1.135632
foo    one       1.212112
       two      -0.173215
qux    one       0.119209
       two      -1.044236
```

## 3. MultiIndex.from_product

```
iterables = [["bar", "baz", "foo", "qux"], ["one", "two"]]
pd.MultiIndex.from_product(iterables, names=["first", "second"])
```

**Output**:

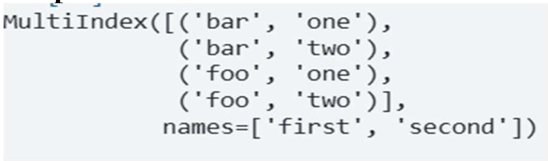


```
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two'),
            ('foo', 'one'),
            ('foo', 'two'),
            ('qux', 'one'),
            ('qux', 'two')],
           names=['first', 'second'])
```

## 4. MultiIndex.from_frame

df = pd.DataFrame(
[["bar", "one"], ["bar", "two"], ["foo", "one"], ["foo", "two"]],
columns=["first", "second"],
)
pd.MultiIndex.from_frame(df)

**Output**:

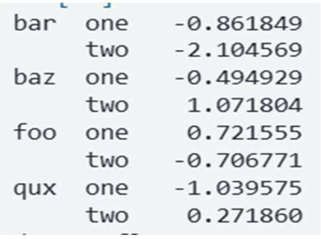


```
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('foo', 'one'),
            ('foo', 'two')],
           names=['first', 'second'])
```

## 5. MultiIndex from Series or Dataframe

arrays = [
    np.array(["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"]),
    np.array(["one", "two", "one", "two", "one", "two", "one", "two"]),
]
s = pd.Series(np.random.randn(8), index=arrays)
print(s)

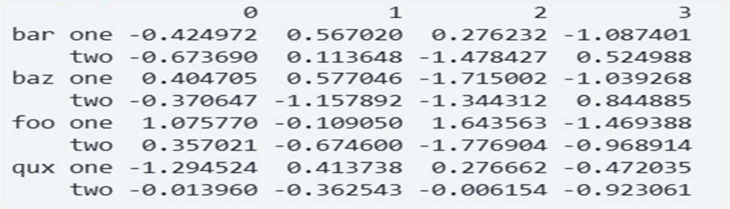
**Output**:



```
bar  one   -0.861849
     two   -2.104569
baz  one   -0.494929
     two    1.071804
foo  one    0.721555
     two   -0.706771
qux  one   -1.039575
     two    0.271860
```

df = pd.DataFrame(np.random.randn(8, 4), index=arrays)
print(df)

**Output**:



```
                0         1         2         3
bar one -0.424972  0.567020  0.276232 -1.087401
    two -0.673690  0.113648 -1.478427  0.524988
baz one  0.404705  0.577046 -1.715002 -1.039268
    two -0.370647 -1.157892 -1.344312  0.844885
foo one  1.075770 -0.109050  1.643563 -1.469388
    two  0.357021 -0.674600 -1.776904 -0.968914
qux one -1.294524  0.413738  0.276662 -0.472035
    two -0.013960 -0.362543 -0.006154 -0.923061
```

# DATA ACQUISITION

## a. Web APIs

An API (application programming interface) is a server that lets you request and send data using code. It serves as a bridge between your application and external systems, allowing you to incorporate valuable data and functionality without having to "manually" get the data yourself.

### How APIs Work

- APIs work by sending requests to a server and receiving responses.
- For instance, when you visit a webpage, your browser makes a request to the server, which sends back the page content. APIs follow the same principle, letting you programmatically retrieve data, access features, or interact with services.

### Why and When to Use an API in Python

- **Real-time data access:** Retrieve up-to-date data on demand, essential for projects that rely on timely information for accurate insights.
- **Access to large datasets**: APIs streamline the integration of vast datasets from multiple sources, eliminating the need for extensive local storage or manual management.
- **Pre-processed insights**: Many APIs supply enriched data, such as sentiment analysis or key entity recognition, reducing the time spent on preprocessing tasks.

## b. Open Data Sources

- **Django REST**: Web browsable APIs and has huge usability for developers, Multiple in-built authentication policies, Serialization which supports both ORM and non-ORM data sources, Extensive and good documentation to refer to and learn, A very active community support, Trusted by organizations like Red Hat, Mozilla, Heroku
- **Flask RESTful**: Flask has a built-in development server and a fast debugger, Flask provides integrated support for unit testing, RESTful request dispatching, Flask support for secure cookies
- **FastAPI**: FastAPI uses advanced features such as async/await syntax and type hints to provide high performance and scalability. Ease to code, Reduced bugs, Asynchronous nature
- **Pyramid**: Pyramid is a Python web framework that is designed to work for making complex, large-scale web applications and APIs. It has various features such as routing, views, authentication, authorization, etc.
- **Falcon**: Falcon is a lightweight high-performance framework that is designed for building fast and lightweight APIs.
- **Bottle**: Bottle is an easy-to-use web framework, which is suitable for small to medium-sized web applications.
- **Eve**: It is designed to be easy to use and flexible, making it an ideal choice for developers who want to build scalable and efficient web applications.
- **Sanic**: Sanic is a Python web framework and it is asynchronous with which the developers can build fast and efficient web applications.
- **Tornado**: Tornado is designed to handle large volumes of concurrent connections. This is known for its high performance and scalability.

## c. Gathering Information from Web APIs

- Requests is one of the most popular python libraries that is not included with python, it has be en proposed that requests be distributed with python by default
- **Python requests is a python library** used to get requests from the web pages
- **Requests** is a Python HTTP library, released under the Apache License 2.0. The goal of the pr oject is to make HTTP requests simpler and more human-friendly. The current version is 2.23. 0.

- **Requests** allows you to send HTTP/1.1 requests extremely easily. There’s no need to manually add query strings to your URLs, or to form-encode your POST data.
- **Human friendly HTTP library**
- Why use Python requests: Donot have to manually add query strings to URLs or form-encode post data

| Method | Description |
|---|---|
| delete(url, args) | Sends a DELETE request to the specified url |
| get(url, params, args) | Sends a GET request to the specified url |
| head(url, args) | Sends a HEAD request to the specified url |
| patch(*url*, *data, args*) | Sends a PATCH request to the specified url |
| post(url, data, json, args) | Sends a POST request to the specified url |
| put(*url*, *data, args*) | Sends a PUT request to the specified url |
| request(*method*, *url*, *args*) | Sends a request of the specified method to the specified url |

## Using requests Library

The requests library is commonly used for interacting with web APIs.

**Example:**

```
pip install requests
```

```
import requests
```

```
r=requests.get("https://www.abc.com/tags/ref_httpmethods.asp")
print(r.text)
```

Output:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
<title>HTTP Methods GET vs POST</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="Keywords" conten
```

```
r=requests.get("https://www.abc.com/tags/ref_httpmethods.asp")
print(r.text)
```

Output:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
<title>HTTP Methods GET vs POST</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="Keywords" content
```

```
payload={'key1':'params'}
r=requests.get("https://www.abc.com/tags/ref_httpmethods.asp",params=payload)
print(r.url)
```

Output:

```
https://www.abc.com/tags/ref_httpmethods.asp?key1=params
```

```
r=requests.get("https://www.abc.com/tags/ref_httpmethods.asp")
print(r.status_code)
```

Output:

```
200
```

```
print(r.cookies)
```

Output: `<RequestsCookieJar[]>`

```
print(r.headers)
```

Output:

```
{'Content-Encoding': 'gzip', 'Accept-Ranges': 'bytes', 'Age': '13070', 'Cache-Control'
: 'Public,public', 'Content-Type': 'text/html', 'Date': 'Tue, 01 Dec 2020 05:20:46 GMT', 'Expi
res': 'Tue, 01 Dec 2020 09:20:46 GMT', 'Last-Modified': 'Tue, 01 Dec 2020 01:42:56 GMT', 'Serv
er': 'ECS (ord/4CB4)', 'Vary': 'Accept-Encoding', 'X-Cache': 'HIT', 'X-Frame-Options': 'SAMEOR
IGIN', 'X-Powered-By': 'ASP.NET', 'Content-Length': '23302'}
```

```
payload={'key1':'value1','key2':'value2'}
r=requests.post('https://www.abc.com/tags/ref_httpmethods.asp/post',data=payload)
print(r.text)
```

Output:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
<title>404 - Page not found</title>
<meta charset="windo
```

```
print(r.url)
```

Output: https://www.abc.com/tags/ref_httpmethods.asp/post

## d. Web scrapping

• Web Scraping is the process of downloading data from webpages and extracting information from that data. It is a great tool to have in your tool kit because it allows you to get rich varieties of data.
• BeautifulSoup is a web scraping library which is best used for small projects.
• Beautiful soup is a library for parsing HTML and XML documents. Requests (handles HTTP sessions and makes HTTP requests) in combination with BeautifulSoup (a parsing library) are the best package tools for small and quick web scraping. For scraping simpler, static, less-JS related complexities, and then this tool is probably what you're looking for.

**Steps involved in web scraping**

- Send an HTTP request to the URL of the webpage you want to access. The server responds to the request by returning the HTML content of the webpage. For this task, we will use a third-party HTTP library for python-requests.
- Once we have accessed the HTML content, we are left with the task of parsing the data. Since most of the HTML data is nested, we cannot extract data simply through string processing. One needs a parser which can create a nested/tree structure of the HTML data. There are many HTML parser libraries available but the most advanced one is html5lib.
- Now, all we need to do is navigating and searching the parse tree that we created, i.e. tree traversal. For this task, we will be using another third-party python library, Beautiful Soup. It is a Python library for pulling data out of HTML and XML files.

### Step 1: Installing the required third-party libraries

```
pip install requests
pip install html5lib
pip install bs4
```

### Step 2: Accessing the HTML content from webpage

```
import requests
URL = "https://www.geeksforgeeks.org/data-structures/"
r = requests.get(URL)
print(r.content)
```

### Step 3: Parsing the HTML content

```
import requests
from bs4 import BeautifulSoup

URL = "http://www.values.com/inspirational-quotes"
r = requests.get(URL)
```

```
soup = BeautifulSoup(r.content, 'html5lib')
print(soup.prettify())
```

In the example above,

```
soup = BeautifulSoup(r.content, 'html5lib')
```

We create a BeautifulSoup object by passing two arguments:

**r.content : It is the raw HTML content.**
**html5lib : Specifying the HTML parser we want to use.**

Example:

```
pip install bs4
import bs4

import urllib.request
sr=urllib.request.urlopen('https://pythonprogramming.net/parsecparseface/').read()
soup=bs4.BeautifulSoup(sr,'lxml')
print(soup.find_all('p'))
```

```
Output: [<p class="introduction">Oh, helloThis page was originally created to help people work with the <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/" target="blank"><strong>Beautiful Soup 4</strong></a> library.</p>, <p> languages</code>:</p>, <p>I think it's clear that, on a scale of 1-10, python is:</p>, <p>Javascript (dynamic data) test:</p>, <p class="jstest" id="yesnojs"> y u bad tho?</p>, <p>Whαţ hαppéns now¿</p>, <p><a href="/sitemap.xml" target="blank"><strong> sitemap</strong></a></p>, <p class="grey-text text-lighten-4">Contactthonprogramming.net.</p>]
```

```
print(soup)
```

```
Output: <html>
<head>
<!--
                    palette:
                    dark blue: #003F72
                    offwhite: #e7d7d7
                    Light Blue: #118AB2
                    Light green: #7DDF64
                    -->
<meta content="width=device-width, initial-scale=1.0" name="viewport"/>
```

```
print(sr)
```

```
Output: b'<html>\n\t<head>\n\n\t\t<!--\n\t\tpalette:\n\t\tdark blue: #003F72\n\t\tyellow: #FFD166\n\t\t
salmon: #EF476F\n\t\toffwhite: #e7d7d7\n\t\tLight Blue: #118AB2\n\t\tLight green: #7DDF64\n\t\t
```

```
print(soup.title)
```

Output: <title>Python Programming Tutorials</title>

```
print(soup.title.name)
```

Output: title

```
print(soup.p.name)
```

Output: p

```
print(soup.title.string)
```

Output: Python Programming Tutorials

```
print(soup.title.text)
Output: Python Programming Tutorials
```

```
print(soup.p)
Output: <p class="introduction">Oh, hello! This is a <span style="font-size:115%">wonderful</span
> page meant to let you practice web scraping. This page was originally created to help people work w
ith the <strong>Beautiful Soup 4</strong></a> library.</p>
```

```
print(soup.find_all('p'))
Output: [<p class="introduction">Oh, hello! This is a <span style="font-size:115%">wonderful</span
> page meant to let you practice web scraping. /BeautifulSoup/bs4/doc/"
```

```
for paragraph in soup.find_all('p'):
    print(paragraph)
Output: <p class="introduction">Oh, hello! This page was originally created to help people work with
the <strong>Beautiful Soup 4</strong></a> library.</p>
<p>The following table gives some general information for programming languages</code>:</p>
<p>I think it's clear that, on a scale of 1-10, python is:</p>
<p>Javascript (dynamic data) test:</p>
```

```
for paragraph in soup.find_all('p'):
    print(paragraph.string)
Output: I think it's clear that, on a scale of 1-10, python is:
Javascript (dynamic data) test:
sitemap
Contact: Harrison@pythonprogramming.net.
Programming is a superpower.
```

```
for paragraph in soup.find_all('p'):
    print(paragraph.text)
Output:    This was originally created to help people work with the Beautiful Soup 4 library.
I think it's clear that, on a scale of 1-10, python is:
Javascript (dynamic data) test:
Contact: Harrison@pythonprogramming.net.
Programming is a superpower.
```

```
print(soup.get_text())
Output:
Python Programming Tutorials
```

```
for url in soup.find_all('a'):
    print(url)
Output:
<a class="brand-logo" href="/"><img class="img-responsive" src="/static/images/mainlogowhitethic
k.jpg" style="width:50px; height;50px; margin-top:5px"/></a>
```

```
for url in soup.find_all('a'):
    print(url.text)
Output: Home
+=1
Support the Content
Community
```

```
for url in soup.find_all('a'):
    print(url.get('href'))
```