Introduction

Pandas is an open source Python package that is most widely used for data science/data analysis and machine learning tasks.

- Provides high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
- Offers powerful, expressive and flexible data structures
- Used in a wide range of fields including academic and commercial domains
- It is built on top of another package named Numpy, which provides support for multi-dimensional arrays.
- Works well with many other data science modules inside the Python ecosystem, and is typically included in every Python distribution

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Data structures in Pandas

It deals with three data structures

1. Series

It is one dimensional array like structure with homogeneous data

Key Features:

- Homogeneous Data
- Size immutable
- Values of data mutable

Example:

The following series is collection of integers

10	23	56	45	76	89	30

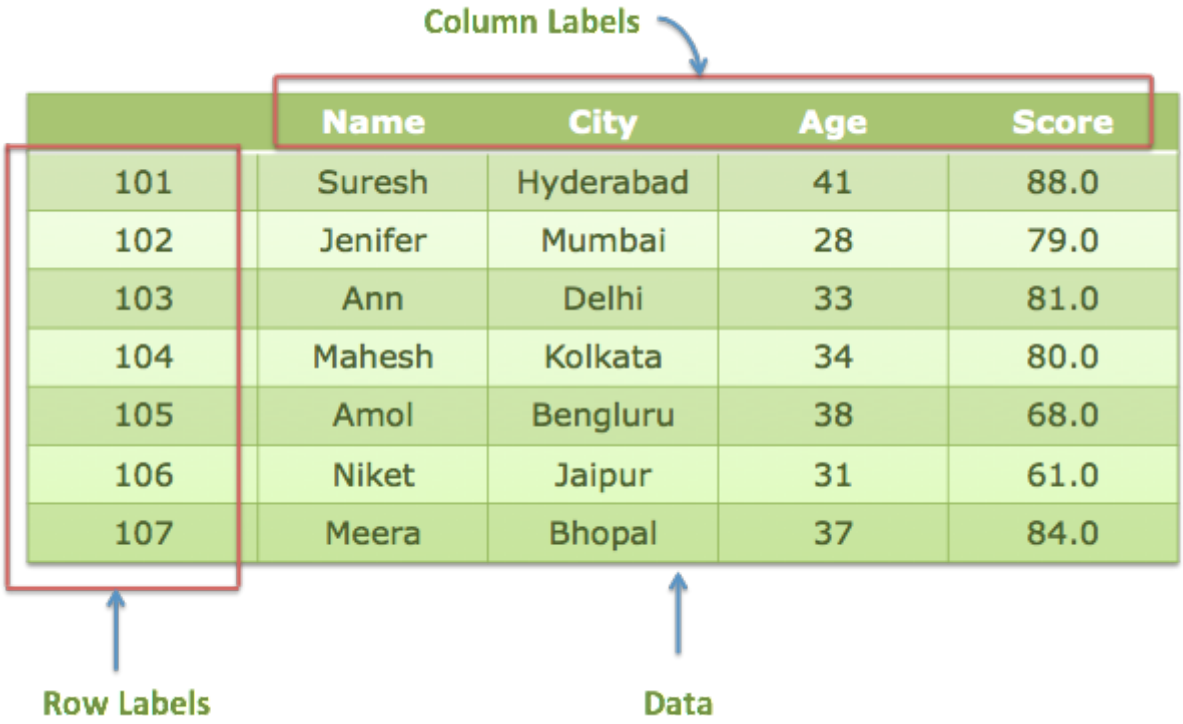
2. DataFrame

It is two dimensional array with heterogeneous data

Key Features:

- Heterogeneous data
- Size Mutable
- Values of data mutable

Example: The table represents details of the candidates and their scores in the exam



Each column represents attribute and row represents each person.

The person can be identified by 'Row Label'

3. Panel

Panel is a three-dimensional data structure with heterogeneous data.

It is hard to represent the panel in graphical representation.

It can be illustrated as a **container of DataFrame.**

Key Features

- Heterogeneous data
- Size Mutable

- Data Mutable

Series in Pandas

Series is a one-dimensional labeled array capable of holding data of any type

It can hold integer, string, float, python objects, etc.

The axis labels are collectively called index.

Series can be created using following constructor

```
pandas.Series(data, index, dtype, copy)
```

Above parameters can be explained as

> **data:** It takes various forms like ndarray, list, constants

> **index:** It must be unique and hashable, same length as data. If no index is passed then by default np.arange(n) is considered

> **dtype:** It represents data type. If not mentioned then data type will be inferred

> **copy:** To create separate copy of data input. Default value is False

Creating Series

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

1. Empty Series

A basic series can be created as empty series

Example 1: Creating empty series in pandas

```
import pandas as p

s = p.Series()

print(s)
```

```
Series([], dtype: float64)
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3: DeprecationWarning: T
  This is separate from the ipykernel package so we can avoid doing imports until
```

2. Creating Series using ndarray

> If data is an ndarray, then index passed must be of the same length.

> If no index is passed, then by default index will be range(n) where n is array length

Program 1: Creating series in pandas using numpy array

```
import pandas as pd
import numpy as np

# Index parameter is not mentioned
data1 = np.array(['a','b','c','d'])
s1 = pd.Series(data1)
print('Series with default Index:')
print(s1)


# Index parameter is mentioned
data2 = np.array(['a','b','c','d'])
s2 = pd.Series(data2,index=[100,101,102,103])
print('Series with specified index:')
print(s2)
```

```
Series with default Index:
0    a
1    b
2    c
3    d
dtype: object
Series with specified index:
100    a
101    b
102    c
103    d
dtype: object
```

In above program,

For series s1, index is not mentioned. hence it takes default values using range function

For Series s2, index is specified. Therefore those values are used

---

3. Creating Series using dictionary

A dict can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index.

If index is passed, the values in data corresponding to the labels in the index will be pulled out.

Program 2: Creating series using dict object without specifying index values

```
import pandas as pd
import numpy as np

# Index parameter is not mentioned
data1 = {'c' : 0.0, 'b' : 1.0, 'a' : 2.0}
s1 = pd.Series(data1)
print('Series with No Specified Index:')
print(s1)

# Index parameter is mentioned
data2 = {'a' : 0.0, 'b' : 1.0, 'c' : 2.0}
s2 = pd.Series(data2,index=['b','c','d','a'])
print('Series with specified Index:')
print(s2)
```

```
Series with No Specified Index:
c    0.0
b    1.0
a    2.0
dtype: float64
Series with specified Index:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

In the above program,

For series s1, index is not specified. Here 'key' elements are considered as indices in sequential order

For series s2, there is no value exist for index 'd'. Hence it is stored as NaN (Not a Number)

---

4. Creating Series using scalar

If data is a scalar value, an index must be provided.

The value will be repeated to match the length of index

Program 3: Creating series using scalar values

```
import pandas as pd
import numpy as np

s = pd.Series(5, index=[0, 1, 2, 3])
print(s)
```

```
0    5
1    5
2    5
3    5
dtype: int64
```

Accessing Data from Series

Data from series can be accessed using

1. Position
2. Label (Index)

1. Accessing data using Position

Data in the series can be accessed similar to that in an ndarray.

All slicing and indexing operations can be applied similarly

Program 4: Python program to demonstrate accessing data from Series using position

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element
print("First Element of Series:",s[0])

#retrieve the first three element
print("First Three Elements:")
print(s[:3])

#retrieve the last three element
print("Last Three Elements:")
print(s[-3:])
```

```
First Element of Series: 1
First Three Elements:
a    1
b    2
c    3
dtype: int64
Last Three Elements:
c    3
d    4
e    5
dtype: int64
```

2. Accessing data using Label (Index)

A Series is like a fixed-size dict in that you can get and set values by index label.

- If single index is specified then single value is retrieved
- If more than one index is specified then multiple values are retrieved. Multiple labels can be passed as **List**
- If specified index(label) is not present then exception is raised

Program 5: Python program to demonstrate accessing data from Series by specifying Labels (Index)

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

# retrieve a single element
print('Single Element:',s['a'])

# retrieve multiple elements
print("Multiple Elements:", s[['a','c','d']])

# specified index is not present
# Following statement will raise an Exception 'KeyError' as label 'f' is not present
#print(s['f'])
```

```
Single Element: 1
Multiple Elements: a    1
c    3
d    4
dtype: int64
```

Operations on Series

Basic arithmetic operations like addition, subtraction, multiplication, and division on two Pandas Series can be performed.

> Perform the required arithmetic operation using the respective arithmetic operator between the two Series

> Result can be assigned the to another Series.

Similarly **Relation Operators** can be used to compare two Series. The result is obtained as a new series with boolean values by element to element comparison

```
# importing the module
import pandas as pd

# creating 2 Pandas Series
s1 = pd.Series([1, 2, 3, 4, 5])
s2 = pd.Series([6, 7, 8, 9, 10])

# Arithmetic operations on the two Series
s3 = s1 + s2
s4 = s1 - s2
s5 = s1 * s2
s6 = s1 / s2

# displaying the result
```

```
print("Addition of two series:")
print(s3)
print("Subtraction of two series:")
print(s4)
print("Multiplication of two series:")
print(s5)
print("Division of two series:")
print(s6)
```

```
Addition of two series:
0     7
1     9
2    11
3    13
4    15
dtype: int64
Subtraction of two series:
0   -5
1   -5
2   -5
3   -5
4   -5
dtype: int64
Multiplication of two series:
0     6
1    14
2    24
3    36
4    50
dtype: int64
Division of two series:
0    0.166667
1    0.285714
2    0.375000
3    0.444444
4    0.500000
dtype: float64
```

```
# importing the module
import pandas as pd

# creating 2 Pandas Series
s1 = pd.Series([1, 2, 3, 4, 5])
s2 = pd.Series([1, 7, 3, 9, 2])

# Comparison between the two Series
s3 = s1 < s2
s4 = s1 <= s2
s5 = s1 > s2
s6 = s1 == s2

# displaying the result
print("Less than relation between two series:")
print(s3)
print("Less than equal to relation between two series:")
print(s4)
print("Greater than relation between two series:")
print(s5)
```

```
print(s5)
print("Equal to relation between two series:")
print(s6)
```

```
Less than relation between two series:
0    False
1     True
2    False
3     True
4    False
dtype: bool
Less than equal to relation between two series:
0     True
1     True
2     True
3     True
4    False
dtype: bool
Greater than relation between two series:
0    False
1    False
2    False
3    False
4     True
dtype: bool
Equal to relation between two series:
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

Dataframe in Pandas

Pandas DataFrames are data structures that contain:

- Data organized in **two dimensions**, rows and columns
- **Labels** that correspond to the **rows and columns**

It can be visualized as a SQL database or Spreadsheet of Excel

DataFrame can be created using following constructor

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

Above parameters can be explained as

> **data:** It takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.

> **index:** It is used for labelling rows in resulting DataFrame. It is Optional. If no index is passed then by default np.arange(n) is considered

> **columns:** It is used for labelling column. It is optional.The default syntax used is - np.arange(n). This is only true if no index is passed.

> **dtype:** It represents data type of each column.

> **copy:** This command is used for copying of data. The default value is False.

Creating DataFrame

A pandas DataFrame can be created using various inputs like

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

1. Creating an empty DataFrame:

A basic DataFrame, which can be created is an Empty Dataframe.

Example 2: Empty Dataframe

```
import pandas as pd

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

```
Empty DataFrame
Columns: []
Index: []
```

2. Creating DataFrame using List

The DataFrame can be created using

- Single list
- List of lists
- List of Dictionary

Program 6: Python program to demonstrate creation of DataFrame using List

```
import pandas as pd

# Creating DataFrame using simple list
data1 = [1,2,3,4,5]
df1 = pd.DataFrame(data1)
print('DataFrame using simple list:')
```

```
print(df1)

print()

# Creating DataFrame using Nested List
data2 = [['Alex',10],['Bob',12],['Clarke',13]]
df2 = pd.DataFrame(data2,columns=['Name','Age'])
print('DataFrame using Nested List')
print(df2)

print()

# Specifying datatype for the columns in DataFrame
df3 = pd.DataFrame(data2,columns=['Name','Age'],dtype=float)
print('DataFrame using Nested List with specified datatype')
print(df3.dtypes)
```

```
DataFrame using simple list:
   0
0  1
1  2
2  3
3  4
4  5

DataFrame using Nested List
     Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13

DataFrame using Nested List with specified datatype
Name     object
Age     float64
dtype: object
```

In List of Dictionaries, key values are taken as column names by default

Program 7: Python program to demonstrate creation of DataFrame using List of Dictionary

```
import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

# Creating DataFrame by passing list of dictionary
df = pd.DataFrame(data)
print('DataFrame with Key values as Column Names')
print(df)

print()

# DataFrame with index specified
df3 = pd.DataFrame(data, index=['first', 'second'])
print('DataFrame with specified Index:')
print(df3)
```

```
print()

# With two column and two indices, column names same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
print('DataFrame considering specified number of columns')
print(df1)

print()

# With two column names and two indices, with one column specifying other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print('DataFrame with one of the specified column name is different')
print(df2)
```

```
DataFrame with Key values as Column Names
   a   b     c
0  1   2   NaN
1  5  10  20.0

DataFrame with specified Index:
        a   b     c
first   1   2   NaN
second  5  10  20.0

DataFrame considering specified number of columns
        a   b
first   1   2
second  5  10

DataFrame with one of the specified column name is different
        a  b1
first   1 NaN
second  5 NaN
```

3. Creating DataFrame using Dictionary

> The keys of the dictionary are the DataFrame's column labels.

> The dictionary values are the data values in the corresponding DataFrame columns.

> The values can be contained in a tuple, list, one-dimensional NumPy array, Pandas Series object, or one of several other data types.

> A single value can be provided as well that will be copied along the entire column

Note:

1. If ndarray is used then all the ndarrays must be of same length.
2. If index is passed, then the length of the index should equal to the length of the arrays.
3. If no index is passed, then by default, index will be range(n), where n is the array length.

Program 8: Python program to demonstrate creation of DataFrame using Dictionary

```
import pandas as pd
import numpy as np

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

# Creating DataFrame using Dictionary with default index
df1 = pd.DataFrame(data)
print('DataFrame with default index')
print(df1)

print()

# Creating DataFrame using Dictionary with specified index
df2 = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print('DataFrame with specified index')
print(df2)

print()

# Creating DataFrame with Dictionary values are of different types
d = {'x': [1, 2, 3], 'y': np.array([2, 4, 8]), 'z': 100}
df3 = pd.DataFrame(d)
print('DataFrame using different type of Dictionary Values')
print(df3)
```

```
DataFrame with default index
    Name  Age
0    Tom   28
1   Jack   34
2  Steve   29
3  Ricky   42

DataFrame with specified index
        Name  Age
rank1    Tom   28
rank2   Jack   34
rank3  Steve   29
rank4  Ricky   42

DataFrame using different type of Dictionary Values
   x  y    z
0  1  2  100
1  2  4  100
2  3  8  100
```

Passing Dictionary of Series:

> Dictionary of Series can be passed to form a DataFrame.

> The resultant index is the union of all the series indexes passed.

Program 9: Python program to create DataFrame using Series

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

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

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
```

In above program series 'one' doesn't have index 'd'.

Therefore while making union of the series it is represented as NaN

4. Creating DataFrame using Files

You can save and load the data and labels from a Pandas DataFrame to and from a number of file types

File format includes:

- CSV
- Excel
- SQL
- JSON and more. This is a very powerful feature.

You can save your job candidate DataFrame to a CSV file with .to_csv( )

Program 10: Python program to demonstrate storing DataFrame in File and creating DataFrame using File

```
# Creating DataFrame
data2 = [['Alex',10],['Bob',12],['Clarke',13]]
df2 = pd.DataFrame(data2,columns=['Name','Age'])
df2.to_csv('Employee.csv')

# Creating DataFrame using Files
w = pd.read_csv('Employee.csv', index_col=0)
print(w)
```

```
     Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13
```

In above method read_csv(), the parameter index_col states that column 0 is representing index for rows

Accessing Data from DataFrame

We are considering data of candidates which appeared for exam.

It includes following attributes:

- Name: name of candidate
- City: City in which he/she has given exam
- Age: Age of the candidate
- Score: Score obtained in exam

Here raw data is considered in form of Dictionary of Lists.

Each key is reprenting column in the DataFrame.

row_labels refers to a list that contains the labels of the rows, which are numbers ranging from 101 to 107.

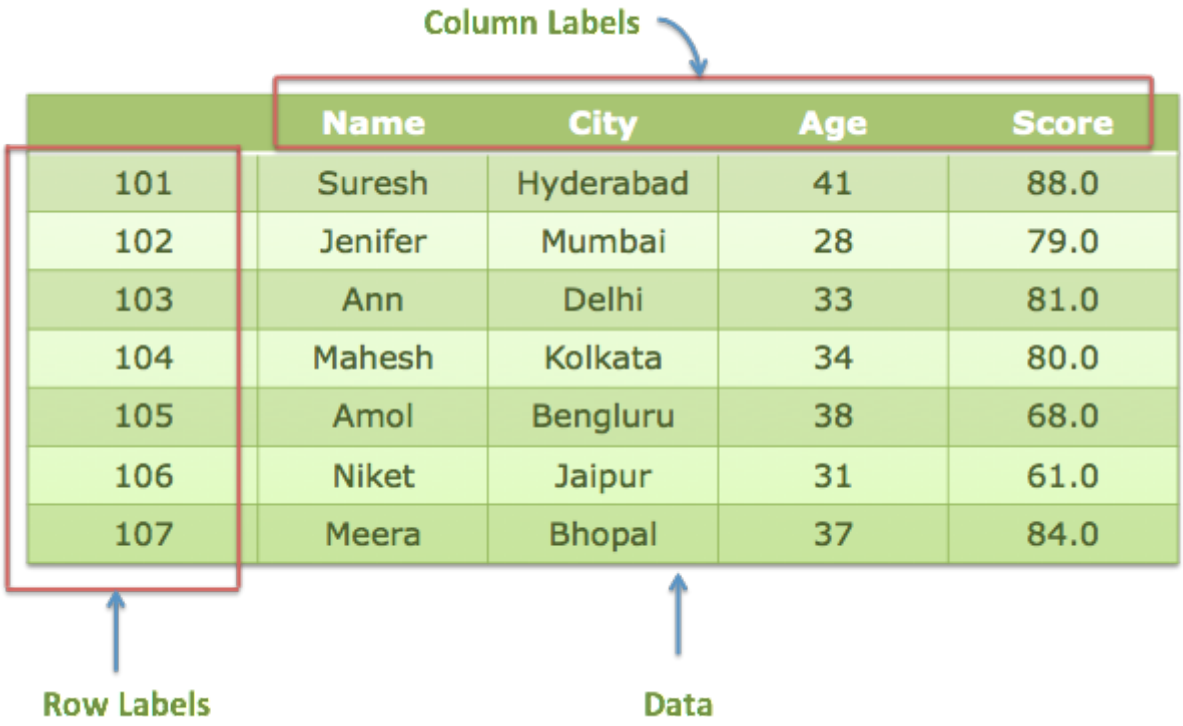
Program 11: Python program to create DataFrame using candidate data appeared for examination

```
import pandas as pd


raw_data = { 'name':['Suresh', 'Jeniffer', 'Ann', 'Mahesh', 'Amol', 'Niket', 'Meera'],
         'city': ['Hyderabad', 'Mumbai', 'Delhi', 'Kolkata', 'Bengluru', 'Jaipur', 'Bhopal
         'age': [41, 28, 33, 34, 38, 31, 37],
         'score': [88.0, 79.0, 81.0, 80.0, 68.0, 61.0, 84.0]
        }

row_labels = [101, 102, 103, 104, 105, 106, 107]

df = pd.DataFrame(data = raw_data, index = row_labels)

print(df)

df.to_csv('Exam_score.csv')
```

```
            name       city  age  score
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
104    Mahesh    Kolkata   34   80.0
105      Amol   Bengluru   38   68.0
106     Niket     Jaipur   31   61.0
107     Meera     Bhopal   37   84.0
```

DataFrame looks just like the candidate table above and has the following features:

- Row labels from 101 to 107
- Column labels such as 'name', 'city', 'age', and 'py-score'
- Data such as candidate names, cities, ages, and Python test scores



Basic Operations on DataFrame:

Some of the methods to access the data:

1. df.head(n=k): It returns top k rows from the DataFrame 'df'

2. df.tail(n=k): It returns bottom k rows from the DataFrame 'df'

3. df.['column_name']: It returns entire column specified in square bracket as pandas Series

4. df.column_name: It also returns entire column specified as an attribute as Pandas series. column_name must be valid identifier in pandas

5. df.loc[n]: It returns entire nth row from DataFrame 'df'

Note: It may be helpful to think of the Pandas DataFrame as a dictionary of columns, or Pandas Series, with many additional features.

Program 12: Program to demonstrate basic operations on DataFrame (Part1)

```
import pandas as pd

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

# Get First three rows from DataFrame
res1 = df.head(n=3)
print('First Three Rows:')
print(res1)

# Get Last three rows from DataFrame
```

```
res2 = df.tail(n=3)
print('Last Three Rows:')
print(res2)
```

```
First Three Rows:
         name       city  age  score
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
Last Three Rows:
      name      city  age  score
105   Amol  Bengluru   38   68.0
106  Niket    Jaipur   31   61.0
107  Meera    Bhopal   37   84.0
```

Program 13: Program to demonstrate basic operations on DataFrame (Part2)

```
import pandas as pd

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

# Retrieve entire column using indexing
res3 = df['name']
print('Column: Name')
print(res3)

# Retrieve entire column using attribute notation
res4 = df.city
print('Column: City')
print(res4)

# Retrieve Entire Row by specifying index
res5 = df.loc[102]
print('Record of Index 102:')
print(res5)

# Retrieve particular city by specifying index
# res4 is representing pandas Series of column City
city = res4[102]
print('City at index 102:', city)
```

```
Column: Name
101      Suresh
102    Jeniffer
103         Ann
104      Mahesh
105        Amol
106       Niket
107       Meera
Name: name, dtype: object
Column: City
101    Hyderabad
102       Mumbai
103        Delhi
```

```
104      Kolkata
105     Bengluru
106       Jaipur
107       Bhopal
Name: city, dtype: object
Record of Index 102:
name     Jeniffer
city       Mumbai
age            28
score          79
Name: 102, dtype: object
City at index 102: Mumbai
```

Retrieving Labels and Data

Pandas DataFrame Labels can be retrieved as sequences using following attributes

1. df.index: It returns row labels of the DataFrame 'df'

2. df.columns: It returns column labels of the DataFrame 'df'

Any individual elements from the above squence can be accessed

Index of the DataFrame can be modified using other sequences or numpy function like arange()

Program: program to retrieve information of the labels

```
import pandas as pd

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

print(df)

print()

# Access Row details
lst = df.index
print("Row Labels")
print(lst)

print()

# Access column details
col = df.columns
print('Column Labels')
print(col)

print()

# Accessing individual elements of column label
attrib = col[2]
print("Name of Column 2:", attrib)
```

```
             name       city  age  score
```

```
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
104    Mahesh    Kolkata   34   80.0
105      Amol   Bengluru   38   68.0
106     Niket     Jaipur   31   61.0
107     Meera     Bhopal   37   84.0

Row Labels
Int64Index([101, 102, 103, 104, 105, 106, 107], dtype='int64')

Column Labels
Index(['name', 'city', 'age', 'score'], dtype='object')

Name of Column 2: age
```

Program: Modifying index of DataFrame

```
import pandas as pd
import numpy as np

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

print('original DataFrame')
print(df)

print()

df.index = np.arange(10,17)
print('DataFrame after modifying row index')
print(df)
```

```
original DataFrame
         name       city  age  score
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
104    Mahesh    Kolkata   34   80.0
105      Amol   Bengluru   38   68.0
106     Niket     Jaipur   31   61.0
107     Meera     Bhopal   37   84.0

DataFrame after modifying row index
        name       city  age  score
10    Suresh  Hyderabad   41   88.0
11  Jeniffer     Mumbai   28   79.0
12       Ann      Delhi   33   81.0
13    Mahesh    Kolkata   34   80.0
14      Amol   Bengluru   38   68.0
15     Niket     Jaipur   31   61.0
16     Meera     Bhopal   37   84.0
```

Data can be extracted from DataFrame without labels using either **to_numpy( )** method or **values** attribute

The Pandas documentation suggests using .to_numpy because of the flexibility offered by two optional parameters:

1. **dtype:** Use this parameter to specify the data type of the resulting array. It's set to None by default.
2. **copy:** Set this parameter to False if we want to use the original data from the DataFrame. Set it to True if we want to make a copy of the data.

Program: To extract the data without labels from DataFrame

```
import pandas as pd
import numpy as np

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

print('original DataFrame')
print(df)

print()

# Extracting Data and storing it in numpy array
df2 = df.to_numpy()
print(df2)
```

```
original DataFrame
         name       city  age  score
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
104    Mahesh    Kolkata   34   80.0
105      Amol   Bengluru   38   68.0
106     Niket     Jaipur   31   61.0
107     Meera     Bhopal   37   84.0

[['Suresh' 'Hyderabad' 41 88.0]
 ['Jeniffer' 'Mumbai' 28 79.0]
 ['Ann' 'Delhi' 33 81.0]
 ['Mahesh' 'Kolkata' 34 80.0]
 ['Amol' 'Bengluru' 38 68.0]
 ['Niket' 'Jaipur' 31 61.0]
 ['Meera' 'Bhopal' 37 84.0]]
```

Retrieving data with accessors

Following accessors can be used to retrieve the data from DataFrame

Accessor	Description
.loc[ ]	• Accepts the labels of rows and columns and returns series of DataFrames. • It can be used to get entire rows or columns as well as their parts
.iloc[ ]	• Accepts zero based indices of rows and columns and returns series of DataFrames. • It can be used to get entire rows or columns as well as their parts
.at[ ]	• Accepts the labels of rows and columns and returns a single data value
.iat[ ]	• Accepts zero based indices of rows and columns and returns a single data value

.loc [ ] and .iloc [ ] functions are very powerful. They can be used for slicing and indexing

Program: To retrieve data using accessors

```
import pandas as pd

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

print(df)
print()

# Access all the rows of column 'city'
df2 = df.loc[:,'city']

print('City column')
print(df2)
print()

# Access all the rows of column 'city' using iloc
df3 = df.iloc[:, 1]

print('Column 1')
print(df3)
print()


# Slicing Operation using loc
df4 = df.loc[101:105, ['name', 'city']]
```

```
print("Name and City columns for First 5 rows")
print(df4)
print()

# Slicing Operation using iloc
df5 = df.iloc[1:6, [0,1] ]

print("Name and City columns for First 5 rows")
print(df5)
print()
```

```
         name       city  age  score
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
104    Mahesh    Kolkata   34   80.0
105      Amol   Bengluru   38   68.0
106     Niket     Jaipur   31   61.0
107     Meera     Bhopal   37   84.0

City column
101    Hyderabad
102       Mumbai
103        Delhi
104      Kolkata
105     Bengluru
106       Jaipur
107       Bhopal
Name: city, dtype: object

Column 1
101    Hyderabad
102       Mumbai
103        Delhi
104      Kolkata
105     Bengluru
106       Jaipur
107       Bhopal
Name: city, dtype: object

Name and City columns for First 5 rows
         name       city
101    Suresh  Hyderabad
102  Jeniffer     Mumbai
103       Ann      Delhi
104    Mahesh    Kolkata
105      Amol   Bengluru

Name and City columns for First 5 rows
         name      city
102  Jeniffer    Mumbai
103       Ann     Delhi
104    Mahesh   Kolkata
105      Amol  Bengluru
106     Niket    Jaipur
```

whenever a single value is needed, Pandas recommends using the specialized accessors .at[ ] and .iat[ ]

```
import pandas as pd

# Loading csv file into DataFrame
df = pd.read_csv('Exam_score.csv', index_col=0)

print(df)
print()

# Accessing value at index 102 and column name
s = df.at[102, 'name']
print(s)

# Accessing value at row 2 and column 0
t = df.iat[2, 0]
print(t)
```

```
          name       city  age  score
101    Suresh  Hyderabad   41   88.0
102  Jeniffer     Mumbai   28   79.0
103       Ann      Delhi   33   81.0
104    Mahesh    Kolkata   34   80.0
105      Amol   Bengluru   38   68.0
106     Niket     Jaipur   31   61.0
107     Meera     Bhopal   37   84.0

Jeniffer
Ann
```

Grouping & Aggregating DataFrames

Any groupby operation involves one of the following operations on the original object. They are −

- Splitting the Object
- Applying a function
- Combining the results

In many situations, the data is splitted into sets and some functionality on each subset may be applied.

In the apply functionality, following operations can be performed −

- Aggregation − computing a summary statistic
- Transformation − perform some group-specific operation
- Filtration − discarding the data with some condition

Split Data into Groups

Pandas object can be split into any of their objects.

There are multiple ways to split an object like −

- obj.groupby('key')
- obj.groupby(['key1','key2'])
- obj.groupby(key,axis=1)

Program : Program to demonstrate group by function

```
import pandas as pd

tennis_data = {'Players': ['Nadal', 'Federer', 'Djokovic', 'Sampras', 'Federer',
   'Nadal', 'Sampras', 'Djokovic', 'Sampras', 'Federer', 'Djokovic', 'Nadal'],
   'Rank': [1, 2, 2, 3, 3, 4 ,1, 1, 2, 4, 1, 2],
   'Year': [2014, 2015, 2014, 2015, 2014, 2015, 2016, 2017, 2016, 2014, 2015, 2017],
   'Points':[876, 789, 863, 673, 741, 812, 756, 788, 694, 701, 804, 690]}
df = pd.DataFrame(tennis_data)

print(df)
print()

# Group by single key
s = df.groupby('Players').groups
print('Grouping data by Player Name:')
print(s)

# Group by multiple keys
t = df.groupby(['Players', 'Year']).groups
print('Grouping data by Player Name and Year:')
print(t)
```

```
     Players  Rank  Year  Points
0      Nadal     1  2014     876
1    Federer     2  2015     789
2   Djokovic     2  2014     863
3    Sampras     3  2015     673
4    Federer     3  2014     741
5      Nadal     4  2015     812
6    Sampras     1  2016     756
7   Djokovic     1  2017     788
8    Sampras     2  2016     694
9    Federer     4  2014     701
10  Djokovic     1  2015     804
11     Nadal     2  2017     690

Grouping data by Player Name:
{'Djokovic': [2, 7, 10], 'Federer': [1, 4, 9], 'Nadal': [0, 5, 11], 'Sampras': [3, 6,
Grouping data by Player Name and Year:
{('Djokovic', 2014): [2], ('Djokovic', 2015): [10], ('Djokovic', 2017): [7], ('Federe
```

---

get_group(): Using this method, we can select a single group

Program : Program to show iteration through groups and use of get_group() method

```
import pandas as pd

tennis_data = {'Players': ['Nadal', 'Federer', 'Djokovic', 'Sampras', 'Federer',
   'Nadal', 'Sampras', 'Djokovic', 'Sampras', 'Federer', 'Djokovic', 'Nadal'],
   'Rank': [1, 2, 2, 3, 3, 4 ,1, 1, 2, 4, 1, 2],
   'Year': [2014, 2015, 2014, 2015, 2014, 2015, 2016, 2017, 2016, 2014, 2015, 2017],
   'Points':[876, 789, 863, 673, 741, 812, 756, 788, 694, 701, 804, 690]}

df = pd.DataFrame(tennis_data)
print(df)

print()

grouped = df.groupby('Year')

# Iteration through groups
for name,group in grouped:
   print(name)
   print(group)

print()
# Selecting a group using method get_group()
df2 = grouped.get_group(2014)
print('Data with Year 2014')
print(df2)
```

```
     Players  Rank  Year  Points
0      Nadal     1  2014     876
1    Federer     2  2015     789
2   Djokovic     2  2014     863
3    Sampras     3  2015     673
4    Federer     3  2014     741
5      Nadal     4  2015     812
6    Sampras     1  2016     756
7   Djokovic     1  2017     788
8    Sampras     2  2016     694
9    Federer     4  2014     701
10  Djokovic     1  2015     804
11     Nadal     2  2017     690

2014
    Players  Rank  Year  Points
0     Nadal     1  2014     876
2  Djokovic     2  2014     863
4   Federer     3  2014     741
9   Federer     4  2014     701
2015
     Players  Rank  Year  Points
1    Federer     2  2015     789
3    Sampras     3  2015     673
5      Nadal     4  2015     812
10  Djokovic     1  2015     804
2016
   Players  Rank  Year  Points
6  Sampras     1  2016     756
8  Sampras     2  2016     694
2017
```

```
     Players  Rank  Year  Points
7   Djokovic     1  2017     788
11     Nadal     2  2017     690

Data with Year 2014
    Players  Rank  Year  Points
0     Nadal     1  2014     876
2  Djokovic     2  2014     863
4   Federer     3  2014     741
9   Federer     4  2014     701
```

Aggregation

An aggregated function returns a single aggregated value for each group.

Once the group by object is created, several aggregation operations can be performed on the grouped data.

Numpy functions can be applied on this group data

Program: Program to demonstrate aggregation operations on the group object

```
import pandas as pd
import numpy as np

tennis_data = {'Players': ['Nadal', 'Federer', 'Djokovic', 'Sampras', 'Federer',
   'Nadal', 'Sampras', 'Djokovic', 'Sampras', 'Federer', 'Djokovic', 'Nadal'],
   'Rank': [1, 2, 2, 3, 3, 4 ,1, 1, 2, 4, 1, 2],
   'Year': [2014, 2015, 2014, 2015, 2014, 2015, 2016, 2017, 2016, 2014, 2015, 2017],
   'Points':[876, 789, 863, 673, 741, 812, 756, 788, 694, 701, 804, 690]}

df = pd.DataFrame(tennis_data)
print(df)

print()

grouped = df.groupby('Year')

# Finding mean of each column(if number) in the group
df2 = grouped.agg(np.mean)
print('Mean of every attribute in the group')
print(df2)

# Finding mean of specific column/attribute
df3 = grouped['Points'].agg(np.mean)
print('Mean of attribute "Points" in the group')
print(df3)

# Finding size of each group
df4 = grouped.agg(np.size)
print('Size of each attribute in the group')
print(df4)
```

```
     Players  Rank  Year  Points
0     Nadal     1  2014     876
```

```
1    Federer     2  2015     789
2   Djokovic     2  2014     863
3    Sampras     3  2015     673
4    Federer     3  2014     741
5      Nadal     4  2015     812
6    Sampras     1  2016     756
7   Djokovic     1  2017     788
8    Sampras     2  2016     694
9    Federer     4  2014     701
10  Djokovic     1  2015     804
11     Nadal     2  2017     690

Mean of every attribute in the group
      Rank  Points
Year
2014   2.5  795.25
2015   2.5  769.50
2016   1.5  725.00
2017   1.5  739.00
Mean of attribute "Points" in the group
Year
2014    795.25
2015    769.50
2016    725.00
2017    739.00
Name: Points, dtype: float64
Size of each attribute in the group
      Players  Rank  Points
Year
2014        4     4       4
2015        4     4       4
2016        2     2       2
2017        2     2       2
```

With grouped Series, a list or dict of functions can be passed to do aggregation with, and generate DataFrame as output

Program:

```
import pandas as pd
import numpy as np

tennis_data = {'Players': ['Nadal', 'Federer', 'Djokovic', 'Sampras', 'Federer',
   'Nadal', 'Sampras', 'Djokovic', 'Sampras', 'Federer', 'Djokovic', 'Nadal'],
   'Rank': [1, 2, 2, 3, 3, 4 ,1, 1, 2, 4, 1, 2],
   'Year': [2014, 2015, 2014, 2015, 2014, 2015, 2016, 2017, 2016, 2014, 2015, 2017],
   'Points':[876, 789, 863, 673, 741, 812, 756, 788, 694, 701, 804, 690]}

df = pd.DataFrame(tennis_data)
print(df)

print()

grouped = df.groupby('Players')

df2 = grouped['Points'].agg([np.sum, np.mean, np.std])
print(df2)
```

```
     Players  Rank  Year  Points
0      Nadal     1  2014     876
1    Federer     2  2015     789
2   Djokovic     2  2014     863
3    Sampras     3  2015     673
4    Federer     3  2014     741
5      Nadal     4  2015     812
6    Sampras     1  2016     756
7   Djokovic     1  2017     788
8    Sampras     2  2016     694
9    Federer     4  2014     701
10  Djokovic     1  2015     804
11     Nadal     2  2017     690
```

```
           sum        mean        std
Players
Djokovic  2455  818.333333  39.501055
Federer   2231  743.666667  44.060564
Nadal     2378  792.666667  94.495150
Sampras   2123  707.666667  43.154760
```

Merging DataFrames

Pandas supports join operations very similar to relational databases like SQL.

Pandas provides a single function, **merge**, as the entry point for all standard database join operations between DataFrame objects

Syntax for merge method on DataFrame is as follows:

```
pd.merge(left, right, how='inner', on=None, left_on=None,
right_on=None, left_index=False, right_index=False, sort=True)
```

The parameters are as follows:

> **left** − A DataFrame object.

> **right** − Another DataFrame object.

> **on** − Columns (names) to join on. Must be found in both the left and right DataFrame objects.

> **how** − One of 'left', 'right', 'outer', 'inner'. Defaults to inner. Each method has been described below.

> **left_on** − Columns from the left DataFrame to use as keys.

> **right_on** − Columns from the right DataFrame to use as keys.

> **left_index** − If True, use the index (row labels) from the left DataFrame as its join key(s).

> **right_index** – If True, use the index (row labels) from the left DataFrame as its join key(s).

> **sort** – Sort the result DataFrame by the join keys in lexicographical order. Default value is set to True.

Program : Program to demonstrate merging of Frames on the basis of id

```
import pandas as pd

data1 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                      'Name': ['Ashutosh', 'Amey', 'Atul', 'Ankur', 'Anvay'],
                      'subject_id':['sub1','sub2','sub4','sub6','sub5']})

data2 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                       'Name': ['Mohan', 'Sameer', 'Vishal', 'Jitesh', 'Hari'],
                       'subject_id':['sub2','sub4','sub3','sub6','sub5']})

print('DataFrame 1:')
print(data1)
print('DataFrame 2:')
print(data2)

# Merging DataFrames on single key
df1 = pd.merge(data1, data2, on='id')
print('Merging DataFrames on id')
print(df1)

# Merging DataFrames on Multiple Keys
df2 = pd.merge(data1, data2, on=['id', 'subject_id'])
print('Merging DataFrames on id and subject id')
print(df2)
```

```
DataFrame 1:
    id      Name subject_id
0  101  Ashutosh       sub1
1  102      Amey       sub2
2  103      Atul       sub4
3  104     Ankur       sub6
4  105     Anvay       sub5
DataFrame 2:
    id    Name subject_id
0  101   Mohan       sub2
1  102  Sameer       sub4
2  103  Vishal       sub3
3  104  Jitesh       sub6
4  105    Hari       sub5
Merging DataFrames on id
    id    Name_x subject_id_x  Name_y subject_id_y
0  101  Ashutosh         sub1   Mohan         sub2
1  102      Amey         sub2  Sameer         sub4
2  103      Atul         sub4  Vishal         sub3
3  104     Ankur         sub6  Jitesh         sub6
```

```
4  105     Anvay          sub5    Hari         sub5
Merging DataFrames on id and subject id
    id Name_x subject_id  Name_y
0  104  Ankur       sub6  Jitesh
1  105  Anvay       sub5    Hari
```

Merge using how argument

The how argument to merge specifies how to determine which keys are to be included in the resulting table.

If a key combination does not appear in either the left or the right tables, the values in the joined table will be NA.

Summary Table:

Merge Method	SQL Equivalent	Description
left	LEFT OUTER JOIN	Use keys from left object
right	RIGHT OUTER JOIN	Use keys from right object
outer	FULL OUTER JOIN	Use Union of keys
inner	INNER JOIN	Use Intersection of keys

Program : Program to demonstrate merging of DataFrames using 'how' argument (Part 1)

```
import pandas as pd

data1 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                       'Name': ['Ashutosh', 'Amey', 'Atul', 'Ankur', 'Anvay'],
                       'subject_id':['sub1','sub2','sub4','sub6','sub5']})

data2 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                        'Name': ['Mohan', 'Sameer', 'Vishal', 'Jitesh', 'Hari'],
                        'subject_id':['sub2','sub4','sub3','sub6','sub5']})

print('DataFrame 1:')
print(data1)
print()
print('DataFrame 2:')
print(data2)

print()

# Merging DataFrame on single key with LEFT JOIN
df1 = pd.merge(data1, data2, on='subject_id', how='left')
print('Merging DataFrames LEFT JOIN operation')
```

```
print('Merging DataFrames LEFT JOIN operation')
print(df1)

print()

# Merging DataFrame on single key with RIGHT JOIN
df2 = pd.merge(data1, data2, on='subject_id', how='right')
print('Merging DataFrames RIGHT JOIN operation')
print(df2)
```

```
DataFrame 1:
    id      Name subject_id
0  101  Ashutosh       sub1
1  102      Amey       sub2
2  103      Atul       sub4
3  104     Ankur       sub6
4  105     Anvay       sub5

DataFrame 2:
    id    Name subject_id
0  101   Mohan       sub2
1  102  Sameer       sub4
2  103  Vishal       sub3
3  104  Jitesh       sub6
4  105    Hari       sub5

Merging DataFrames LEFT JOIN operation
   id_x    Name_x subject_id   id_y  Name_y
0   101  Ashutosh       sub1    NaN     NaN
1   102      Amey       sub2  101.0   Mohan
2   103      Atul       sub4  102.0  Sameer
3   104     Ankur       sub6  104.0  Jitesh
4   105     Anvay       sub5  105.0    Hari

Merging DataFrames RIGHT JOIN operation
    id_x Name_x subject_id  id_y  Name_y
0  102.0   Amey       sub2   101   Mohan
1  103.0   Atul       sub4   102  Sameer
2    NaN    NaN       sub3   103  Vishal
3  104.0  Ankur       sub6   104  Jitesh
4  105.0  Anvay       sub5   105    Hari
```

Program : Program to demonstrate merging of DataFrames using 'how' argument (Part 2)

```
import pandas as pd

data1 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                       'Name': ['Ashutosh', 'Amey', 'Atul', 'Ankur', 'Anvay'],
                       'subject_id':['sub1','sub2','sub4','sub6','sub5']})

data2 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                        'Name': ['Mohan', 'Sameer', 'Vishal', 'Jitesh', 'Hari'],
                        'subject_id':['sub2','sub4','sub3','sub6','sub5']})

print('DataFrame 1:')
print(data1)
print()
```

```
print()
print('DataFrame 2:')
print(data2)

# Merging DataFrame on single key with OUTER JOIN
df3 = pd.merge(data1, data2, on='subject_id', how='outer')
print('Merging DataFrames OUTER JOIN operation')
print(df3)

print()

# Merging DataFrame on single key with INNER JOIN
df4 = pd.merge(data1, data2, on='subject_id', how='inner')
print('Merging DataFrames INNER JOIN operation')
print(df4)
```

Note:

> Joining will be performed on index.

> Join operation honors the object on which it is called. So, a.join(b) is not equal to b.join(a).

Program : Program to demonstrate Join operations 'a on b' is not equal to 'b on a'

```
import pandas as pd

data1 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                      'Name': ['Ashutosh', 'Amey', 'Atul', 'Ankur', 'Anvay'],
                      'subject_id':['sub1','sub2','sub4','sub6','sub5']})

data2 = pd.DataFrame({ 'id':[101, 102, 103, 104, 105],
                       'Name': ['Mohan', 'Sameer', 'Vishal', 'Jitesh', 'Hari'],
                       'subject_id':['sub2','sub4','sub3','sub6','sub5']})

print('DataFrame 1:')
print(data1)
print()
print('DataFrame 2:')
print(data2)

print()

# LEFT Join operation data1 on data2
df1 = pd.merge(data1, data2, on='subject_id', how='left')
print('LEFT JOIN operation data1 on data2')
print(df1)

print()

# Left Join operstion data2 on data1
df2 = pd.merge(data2, data1, on='subject_id', how='left')
print('LEFT JOIN operation data2 on data1')
print(df2)
```

```
print(df2)
```

```
DataFrame 1:
    id      Name subject_id
0  101  Ashutosh       sub1
1  102      Amey       sub2
2  103      Atul       sub4
3  104     Ankur       sub6
4  105     Anvay       sub5

DataFrame 2:
    id    Name subject_id
0  101   Mohan       sub2
1  102  Sameer       sub4
2  103  Vishal       sub3
3  104  Jitesh       sub6
4  105    Hari       sub5

LEFT JOIN operation data1 on data2
   id_x    Name_x subject_id   id_y  Name_y
0   101  Ashutosh       sub1    NaN     NaN
1   102      Amey       sub2  101.0   Mohan
2   103      Atul       sub4  102.0  Sameer
3   104     Ankur       sub6  104.0  Jitesh
4   105     Anvay       sub5  105.0    Hari

LEFT JOIN operation data2 on data1
   id_x  Name_x subject_id   id_y Name_y
0   101   Mohan       sub2  102.0   Amey
1   102  Sameer       sub4  103.0   Atul
2   103  Vishal       sub3    NaN    NaN
3   104  Jitesh       sub6  104.0  Ankur
4   105    Hari       sub5  105.0  Anvay
```