Python Programming

Pandas

Agenda :

- Introduction
- Installation
- Data Structure in Pandas
- Series
- Dataframe

Introduction :

PANDAS (PANel DAta) is a high-level data manipulation tool **used for analysing data.**

Fast and efficient for manipulating and analyzing data.

Easy handling of missing data.

It is very easy to import and export data using Pandas library which has a very rich set of functions.

Pandas has three important data structures, namely – Series, DataFrame and Panel to make the process of analysing data organised, effective and efficient.

Installing Pandas :

To install Pandas from command line, we need to type in:

```
pip install pandas
```

Data Structure in Pandas :

A data structure is a collection of data values and operations that can be applied to that data.

It enables efficient storage, retrieval and modification to the data.

Two commonly used data structures in Pandas are:

- Series
- DataFrame

Series :

A Series is a **one-dimensional array** containing a sequence of values of **any data type** (int, float, list, string, etc) which by default have numeric data labels starting from zero.

The data label associated with a particular value is called its index.

We can also assign values of other data types as index.

We can imagine a Pandas Series as a column in a spreadsheet.

Index	Value
0	Arnab
1	Samridhi
2	Ramit
3	Divyam
4	Kritika

Creation of Series :

1. **Creation of Series from Scalar Values :**

```
>>> series1 = pd.Series([10,20,30]) #create a Series

>>> print(series1) #Display the series
```

Output:

```
0      10
1      20
2      30
dtype: int64
```

Output is shown in two columns - the index is on the left and the data value is on the right. by default indices range from 0 through N – 1.

We can also assign user-defined labels to the index and use them to access elements of a Series.

```
>>> series2 = pd.Series(["Kavi","Shyam","Ra vi"], index=[3,5,1])
```

Output:

```
3        Kavi
5        Shyam
1        Ravi
dtype: object
```

We can also use letters or strings as indices.

```
>>> series2 = pd.Series([2,3,4],index=["Feb","M ar","Apr"])
```

2. Creation of Series from NumPy Arrays :

```
>>> array1 = np.array([1,2,3,4])

>>> series3 = pd.Series(array1)

>>> print(series3)
```

Output:

```
0     1
1     2
2     3
3     4
dtype: int32
```

3. Creation of Series from Dictionary :

```
>>> dict1 = {'India': 'NewDelhi', 'UK': 'London', 'Japan': 'Tokyo'}

>>> series8 = pd.Series(dict1)

>>> print(series8)

    #Display the series
    India NewDelhi
    UK London
    Japan Tokyo
    dtype: object
```

Accessing Elements of a Series :

Indexing :

Indexing Indexing in Series is similar to that for NumPy arrays.

```
>>> seriesMnths = pd.Series([2,3,4],index=["Feb ","Mar","Apr"])
>>> seriesMnths["Mar"]
3
```

More than one element of a series can be accessed using a list of positional integers or a list of index labels.

```
>>> seriesCapCntry = pd.Series(['NewDelhi', 'WashingtonDC', 'London', 'Paris'],
index=['India', 'USA', 'UK', 'France'])
>>> seriesCapCntry[[3,2]]
```

The index values associated with the series can be altered by assigning new index values.

```
>>> seriesCapCntry.index=[10,20,30,40]

>>> seriesCapCntry
    10   NewDelhi
    20   WashingtonDC
    30   London
    40   Paris
```

Slicing :

```
>>> seriesCapCntry = pd.Series(['NewDelhi', 'WashingtonDC', 'London', 'Paris'],
index=['India', 'USA', 'UK', 'France'])
```

```
>>> seriesCapCntry[1:3] #excludes the value at index position 3
    USA       WashingtonDC
    UK        London
    dtype: object
```

```
>>> seriesCapCntry['USA' : 'France']
```

If labelled indexes are used for slicing, then value at the end index label is also included in the output.

We can also get the series in reverse order, for example:

```
>>> seriesCapCntry[ : : -1]
```

We can also use slicing to modify the values of series elements.

```
>>> seriesAlph = pd.Series(np.arange(10,16,1), index = ['a', 'b', 'c', 'd', 'e', 'f'])
>>> seriesAlph[1:3] = 50
>>> seriesAlph
    a     10
    b     50
    c     50
    d     13
    e     14
    f     15
    dtype: int32
```

Attributes of Series :

Series.name - assigns a name to the Series

Series.index.name - assigns a name to the index of the series

Series.values - prints a list of the values in the series

Series.size - prints the number of values in the Series object

Series.empty - prints True if the series is empty, and False otherwise

Methods of Series :

head(n) - Returns the first n members of the series. If the value for n is not passed, then **by default n takes 5** and the first five members are displayed.

count() - Returns the number of non-NaN values in the Series

tail(n) - Returns the last n members of the series. If the value for n is not passed, then by default n takes 5 and the last five members are displayed.

Mathematical Operations on Series :

Addition of two Series :

Add two series using addition operator. The output of addition is NaN if one of the elements or both elements have no value.

```
>>> seriesA + seriesB
```

Add series using add function. This method is applied when we do not want to have NaN values in the output.

```
>>> seriesA.add(seriesB, fill_value=0)
```

Subtraction :

1.seriesA – seriesB

2.seriesA.sub(seriesB, fill_value=1000)

Multiplication of two Series :

1.seriesA * seriesB

2.seriesA.mul(seriesB, fill_value=0)

Division of two Series:

seriesA/seriesB

2.seriesA.divide(seriesB, fill_values=0)

DataFrame :

A DataFrame is a two-dimensional labelled data structure like a table of MySQL.

It contains rows and columns, and therefore has both a row and column index.

Each column can have a different type of value such as numeric, string, boolean, etc., as in tables of a database.

Creation of DataFrame :

Creation of an empty DataFrame :

```
>>>dFrameEmt = pd.DataFrame()

>>> dFrameEmt

Empty DataFrame

Columns: []

Index: []
```

Creation of DataFrame from NumPy ndarrays :

```
dFrame5 = pd.DataFrame([array1, array3, array2], columns=[ 'A', 'B', 'C', 'D'])
>>> dFrame5
    A   B   C   D
0   10  20  30  NaN
1   -10 -20 -30 -40.0
2   100 200 300 NaN
```

Creation of DataFrame from List of Dictionaries :

- We can create DataFrame from a list of Dictionaries, for example:

```
>>> listDict = [{'a':10, 'b':20}, {'a':5, 'b':10, 'c':20}]

>>> dFrameListDict = pd.DataFrame(listDict)

>>> dFrameListDict
    a   b   c
0   10  20  NaN
1   5   10  20.0
```

- Here, the dictionary keys are taken as column labels, and the values corresponding to each key are taken as rows.(rows = no of dicts)
- Number of columns = maximum number of keys in any dictionary of the list

Creation of DataFrame from Dictionary of Lists :

```
>>> dictForest = {'State': ['Assam', 'Delhi', 'Kerala'], 'GArea': [78438, 1483,
38852] , 'VDF' : [2797, 6.72,1663]}

>>> dFrameForest= pd.DataFrame(dictForest)

>>> dFrameForest
     State    GArea    VDF
0    Assam    78438    2797.00
1    Delhi    1483     6.72
2    Kerala   38852    1663.00
```

dictionary keys become column labels by default in a DataFrame, and the lists become the rows.

We can **change the sequence of columns** in a DataFrame.

This can be done by assigning a particular sequence of the dictionary keys as columns parameter, for example:

```
>>> dFrameForest1 = pd.DataFrame(dictForest, columns = ['State','VDF',
'GArea'])

>>> dFrameForest1
     State    VDF        GArea
0    Assam    2797.00    78438
1    Delhi    6.72       1483
2    Kerala   1663.00    38852
```

Creation of DataFrame from Series :

```
dFrame7 = pd.DataFrame([seriesA, seriesB])

>>> dFrame7
     a      b      c      d      e
0    1      2      3      4      5
1    1000   2000   -1000  -5000  1000
```

Creation of DataFrame from Dictionary of Series:

```
>>>ResultDF = pd.DataFrame(ResultSheet)
```

Here resultSheet is a dictionary of series.

Operations on rows and columns in DataFrames :

Adding a New Column to a DataFrame:

```
>>> ResultDF['Preeti']=[89,78,76]
```

Assigning values to a new column label that does not exist will create a new column at the end.

If the column already exists in the DataFrame then the assignment statement will update the values of the already existing column.

Adding a New Row to a DataFrame :

We can add a new row to a DataFrame using the DataFrame.loc[ ] method.

ResultDF.loc['English'] = [85, 86, 83, 80, 90, 89]

We cannot use this method to add a row of data with already existing (duplicate) index value (label).

DataFRame.loc[] method can also be used to change the data values of a row to a particular value.

Further, we can set all values of a DataFrame to a particular value, for example:

```
>>> ResultDF[: ] = 0
```

Deleting Rows or Columns from a DataFrame :

We can use the DataFrame.drop() method to delete rows and columns from a DataFrame.

We need to specify the names of the labels to be dropped and the axis from which they need to be dropped.

To delete a row, the parameter axis is assigned the value 0 and for deleting a column,the parameter axis is assigned the value 1.

```
>>> ResultDF = ResultDF.drop('Science', axis=0)
```

If the DataFrame has more than one row with the same label, the DataFrame.drop() method will delete all the matching rows from it.

Renaming Row Labels of a DataFrame :

We can change the labels of rows and columns in a DataFrame using the DataFrame.rename() method.

```
>>> ResultDF=ResultDF.rename({'Maths':'Sub1', ‘Science':'Sub2','English':'Sub3', 'Hindi':'Sub4'}, axis='index')
```

The parameter axis='index' is used to specify that the row label is to be changed. If no new label is passed corresponding to an existing label, the existing row label is left as it is.

Renaming Column Labels of a DataFrame :

To alter the column names we can again use the rename() method, as shown below.

The parameter axis='columns' implies we want to change the column labels:

```
>>> ResultDF=ResultDF.rename({'Arnab':'Student1','Ramit':'Student2','
Samridhi':'Student3','Mallika':'Student4'},axis='columns')
```

Accessing DataFrames Element through Indexing :

Label Based Indexing :

There are several methods in Pandas to implement label based indexing.

DataFrame.loc[ ] is an important method that is used for label based indexing with DataFrames.

```
>>> ResultDF.loc['Science']
```

When a single column label is passed, it returns the column as a Series.

```
>>> ResultDF.loc[:,'Arnab']
```

To read more than one row from a DataFrame, a list of row labels is used as shown below. Note that using [[]] returns a DataFrame.

Boolean Indexing :

In Boolean indexing, we can select the subsets of data based on the actual values in the DataFrame rather than their row/column labels.

```
>>> ResultDF.loc['Maths'] > 90
```

Accessing DataFrames Element through Slicing :

We can use slicing to select a subset of rows and/or columns from a DataFrame.

```
>>> ResultDF.loc['Maths': 'Science']
```

```
>>> ResultDF.loc['Maths': 'Science', ‘Arnab’]
```

We may use a slice of labels with a slice of column names to access values of those rows and columns:

>>> We may use a slice of labels with a slice of column names to access values of those rows and columns.

Filtering Rows in DataFrames :

In DataFrames, Boolean values like True (1) and False (0) can be associated with indices. They can also be used to filter the records using the DataFrmae.loc[] method.

```
>>> ResultDF.loc[[True, False, True]]
```

Joining of DataFrames :

Joining :

We can use the pandas.DataFrame.append() method to merge two DataFrames.It appends rowsof the second DataFrame at the end of the first DataFrame.

Use pandas.concat([dataFrame1, DataFrame2]) for newer versions of pandas. https://pandas.pydata.org/docs/reference/api/pandas.concat.html

Columns not present in the first DataFrame are added as new columns.

To get the column labels appear in sorted order we can set the parameter sort=True.

Attributes of DataFrames :

DataFrame.index - to display row labels

DataFrame.columns - to display column labels

DataFrame.dtypes - to display data type of each column in the DataFrame

DataFrame.values - to display a NumPy ndarray having all the values in the DataFrame, without the axes labels.

DataFrame.shape - to display a tuple representing the dimensionality of the DataFrame

DataFrame.size - to get number valuse of the DataFrame

DataFrame.T - to transpose the DataFrame. Means, row indices and column labels of the DataFrame replace each other’s position

DataFrame.head(n) - to display the first n rows in the DataFrame

DataFrame.tail(n) - to display the last n rows in the DataFrame

DataFrame.empty - to return the value True if DataFrame is empty and False otherwise

Importing and Exporting Data from CSV Files and DataFrames :

Importing a CSV file to a DataFrame :

We can load the data from the ResultData.csv file into a DataFrame, say marks using Pandas read_csv().

```
>>> marks = pd.read_csv(r"C:\Users\atalb\Documents\Python training notebooks\ResultData.csv",sep =",", header=0)
```

Parameter header specifies the number of the row whose values are to be used as the column names.

Exporting a DataFrame to a CSV file :

We can use the to_csv() function to save a DataFrame to a text or csv file.

```
>>> ResultDF.to_csv(path_or_buf=r"C:\Users\atalb\Documents\Python training notebooks\ResultData_out.csv", sep=',')
```

Parameter index=False is used when we do not want the row labels to be written to the file on disk.

References and Links :

Python Documentation : https://docs.python.org/3/

Pandas Documentation : https://pandas.pydata.org/docs/

GitHub link : https://github.com/lunatic-bot/PythonTraining

Thank You