A guide to Python Pandas Library – by Malay Mitra

'pandas' is a high-performance library in Python that provides a comprehensive set of data structures for manipulating tabular data. There are three primary data structures in pandas - the Series , the DataFrame and Panel objects.

Pandas deals with the following three data structures −

- Series (1D) - 1D labeled homogeneous array, size immutable.
- DataFrame (2D) - 2D labeled, size-mutable tabular structure with heterogeneous columns
- Panel (3D) - General 3D labeled, size-mutable array

Mutability

All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.

Note − DataFrame is widely used and one of the most important data structures.

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

10	23	56	17	52	61	73	90	26	72

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

DataFrame

DataFrame is a two-dimensional array with heterogeneous data. For example,

Name	Age	Gender	Rating
Sanjib	32	Male	3.45
Animesh	28	Male	4.6
Dipak	45	Male	3.9
Aniket	38	Male	2.78

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Creating Series:

A 'pandas' series can be created by

- a NumPy ndarray
- a scalar value
- a Python list / [ ]
- a Python Dictionary / { }

Note :Before using pandas one has to import the following Python libraries for all the below mentioned examples.

```
import numpy as np              # Importing Numpy library
import pandas as pd             # Importing Pandas library
from numpy.random import randint
```

Example - 1

```
s1 = pd.Series(2)
print(s1)                 # Output shows index and value along with dtype=int64
```

Example - 2

```
# Create a series from List
s2 = pd.Series([1,2,3,4,5])
print(s2)                 # Shows index and values
print(s2.values)          # Using attributes 'values'  . Print only values
print(s2.index)           # Print only indexes
```

Example - 3

```
# Using index parameter as constructor
s3 = pd.Series([1,2,3], index=['a', 'b','c'])
print(s3)                 # Print values as 1, 2,3 and indexes as 'a', 'b', 'c'
print(s3.index)           # Index(['a', 'b', 'c'], dtype='Object')
print(s3['c'])            # lookup for a value for index 'c' (3)
# Now change the value to 100 at index='a'
s3['a']=100
print(s3)                 # Values 100,2,3
```

Example – 4

```
# Generate a Series from 5 normal random numbers.
# Combined usage of numpy & pandas
s4 = pd.Series(np.random.randint(low=0, high=10, size=5))
print(s4)                 # Prints 5 random nos. against default index 0 to 4
```

Example – 5

```
# Use np.arange() for Series
s5 = pd.Series(np.arange(0,9))      # np.arange gives array([0, 1, 2, 3, 4, 5, 6, 7,8])
print(s5)                           # Values 0,1,2,...8, index by default 0,1,2,.....8
```

Example – 6

```
# Specify dtype/index
s6 = pd.Series([4, 7, -5,3], index=['d','b', 'a', 'c'], dtype='int32')
print(s6)          # Output index as d,b,a,c with values=4,7,-5,3
```

Example – 7

```
# Create Series from Python dictionary
s7 = pd.Series({ 'a' :1, 'b' :2, 'c':3, 'd' : 4})
print(s7)          # Index takes 'a', 'b', 'c', 'd'. Values are 1,2,3,4
```

Example – 8

```
# Series with NaN / Uniqueness / Size/ Shape / Count
# 'NaN' ? Not a Number data type in Numpy
s8 = pd.Series([1,4,2,6,6,np.NaN])
print(len(s8))              # Length of the series
print(s8.count())           # Count number of non NaN values.5
print(s8.unique())          # Print Unique value in a series
# Count of non-NaN values
print(s8.value_counts())
```

Example - 9

```
# When a key not found – Pandas series treat that as 'NaN'
sdata = {'Kolkata':6700, 'Delhi':6900, 'Mumbai':6750, 'Bangaluru':4500, 'Guwahati':5900,
'Patna':5800, 'Kochi':6000}
states = ['Kolkata', 'Chennai','Delhi', 'Mumbai', 'Kochi', 'Guwahati', 'Patna', 'Bangaluru']
s9 = pd.Series(sdata, index=states)
print(s9)             # Prints 'NaN' for Chennai as it does not exist in sdata
# 'isnull' and 'notnull' instances are used to detect missing data
print(pd.isnull(s9))    # True for Chennai, False for others

# Peeking  and taking specific items in series
print(s9.head())          # head() returns first 5. head(3) returns first 3
print(s9.tail())          # tail() returns last 5, tail(3) returns last 3
print(s9.take([0,2,3]))   # take items 0,2,3

# Use [ ] To search by index label or position
print('Kolkata :', s9['Kolkata'])
print('Kochi :', s9['Kochi'])
```

Example – 10

```
# Multiply a Series by a scalar
s3 = pd.Series([1,2,3,4], index=['a', 'b', 'c', 'd'])
print(s3 * 3)              # Output 3,6,9,12
```

- **<u>Case – where index labels are integers but not starting from 0 and usage of 'loc' and 'iloc'</u>**

  loc → To lookup based on Index label. **Label Based**

  iloc → To lookup by position. 'I' stands for integer. **Position / Location based**

Example – 11

```
ser1  = pd.Series([1,2,3], index=[10,11,12])
print(ser1[10])           # outputs 1. Index label passed an integer
                          # But ser1[0] would raise Exception as there is no index [0]
```

- **Alignment in Pandas Series via Index labels and Arithmetic Operation on Series (+, -, *, /)**

Example – 12

```
# Example of automatic alignment of two series by their
# index label when they are added
s1 = pd.Series([1,2,3,4], index=['a', 'b', 'c','d'])
s2 = pd.Series([4,3,2,1], index=['d', 'c', 'b', 'a'])
# Now add them to find that the values are properly added based on the index labels
print(s1+s2)
print(s1-s2)      # All values are zeros
print(s1 * s2)    # Align and multiply
print(s1/s2)      # Align and divide
```

Remember : Alignment is performed in Arithmetic operation across two Series
While doing alignment, non-intersecting labels appear as 'NaN'.

Example – 13

```
first = pd.Series ({'a':1, 'b':2, 'c':3, 'd':4})
sec = pd.Series({'b':6, 'c':7, 'd':9, 'e':10})
# 'a' and 'e' are not-intersecting
print(first+sec)          # 'a' and 'e' appear as NaN
```

Example – 14

```
# Arithmetic operation with duplicate indexes
s1=pd.Series([1,2,3], index=['a','a','e'])
s2=pd.Series([4,5,6], index=['a','a','c'])
print(s1+s2)              # Output shows Index : a,a,a,a,c,e/ Values : 5,6,6,7,NaN,NaN
# There are 4 nos of 'a' as It makes Cartesian products of the labels
```

Special case of NaN

Pandas handle NaN in a special manner (Compared to Numpy) and does not break the computations like Numpy. Pandas is lenient with missing data assuming it is a common situation in Data analysis

Example - 15

In Numpy

```
nda1= np.array([1,2,3,4,np.NaN])
nda1.mean()               # outputs NaN
```

In Pandas

```
s=pd.Series(nda1)
s.mean()                          # output 2.5, ignoring NaN
#To handle NaN in Pandas like Numpy, use skipna=False
s.mean(skipna=False)              # Outputs NaN
```

- **Boolean Selection on Pandas Series**

To select which items have value > 5 but it does not modify the original Series

Example – 16

```
s = pd.Series([2, 5 , 7, 3, 10])
print(s>5)          # Outputs in terms of True/False with dtype=bool
# Another version to output index & values
print(s[s>5])       # Outputs 7, 10
# Multiple logical conditions
print(s[ (s > 5) & (s < 8)])    # Output 7
```

 # any() and all() method

```
# any() Returns True if any one value in the Series satisfy the conditions
# all() Returns True if all elements satisfy the conditions
print((s>5).all())          # False - since all elements are not > 5
print((s>5).any())          # True - since any one of the elements satisfy the conditions
print((s>5).sum())          # Only 7, 10. Number of items > 5. Output 2.
```

- **Slicing Series [start:stop:step] like we did in normal Python**

Example – 17

```
# Slice Series
s = pd.Series([2,4,6,8,10])         # Index becomes = 0,1,2,3,4
print(s[1:3])                       # Access  index 1,2 but not 3. Prints 4,6
print(s[0:4:2])                     # Index Starts from 0, stops before 4, step 2. Output 2, 6
print(s[:3])                        # start not mentioned so default 0, stops before 3
                                    # So access values at index 0,1,2 with Values=2,4,6
print(s[1:])                        # From 1st to end, output 4, 6, 8, 10
print(s[::-1])                      # Reverse the Series. –
print(s[-3:])                       # Last three.  6, 8, 10
```

 # Slicing with index labels like 'a', 'b','c','d'

```
s=pd.Series([1,2,3,4], index=['a','b','c','d'])
print(s['b':'d'])       # Here it starts from index 'b' and ends at index 'd' (not like normal slicing)
                        # Prints Values : 2,3,4
```

- **Note: Modification via slice would also affect the original Series**

```
slice=s[:2]                 # Takes 0, 1
slice[0]=99
print(s)                    # Value at index 0 changed to 99
```

Pandas DataFrame (DF) Object

- **Create an Empty DF**

```
df = pd.DataFrame()
print(df)                    # Prints the empty DF
```

- **Create a DF from List**

```
data = [2,4,6,8,10]        # List
df = pd.DataFrame(data)
print(df)          # Index appears as 0,1,2,3,4. Values are 2,4,6,8,10 with col heading as 0

data=[ ['Ajay', 10], ['Saurav', 15], ['Dipak', 18]]
df = pd.DataFrame(data, columns=['Name', 'Age'])
print(df)
# DF from List with 'dtype' mentioned
data = [ ['Ajay', 10], ['Sanjib', 15], ['Dipak', 17] ]
df = pd.DataFrame(data, columns=['Name', 'Age'], dtype=float)
print(df)                  # Age appears in float, Index comes as 0,1,2
```

- **Create a DF from dictionary and indexes**

```
data = { 'Name': ['Abhijit', 'Saurav', 'Aditya'],  'Age': [20, 25, 26] }
df=pd.DataFrame(data)
print(df)
# Create a DF with user defined indexes
data = { 'Name': ['Abhijit', 'Saurav', 'Aditya'],  'Age': [20, 25, 26] }
df=pd.DataFrame(data, index=['stud1', 'stud2', 'stud3'])
print(df)
# Change Indexes
df.index = ['A001', 'A002', 'A003']
print(df)
```

- **Create a DF from Numpy 'ndarray'**

```
# Create a DF from 2D ndarray
df = pd.DataFrame(np.array([[10,11], [21,22]]))
print(df)

# Dimension of a DF as determined by '.shape' property, is always 2D
print(df.shape)    # Here it is 2x2, 2 rows, 2 columns
```

- **Specify columns names by 'columns' parameter.**

```
df=pd.DataFrame(np.array([[10,11],[20,21]]), columns=['a', 'b'])
print(df)                          # Columns become as 'a' and 'b' and Index=0,1
print(df.columns)
```

- **Renaming columns in DFs**

```
df.columns = ['c1', 'c2']
print(df)                          # Output shows 'c1' and 'c2' as column heading
```

- **Create a DF with named columns and rows/index**

```
df=pd.DataFrame(np.array( [[10,11], [12,13]]), columns = ['c1','c2'], index=['r1', 'r2'])
print(df)
print(df.index)
print(df.columns)
```

- **Pandas DataFrame (DF) with Data Files (.csv format)**

  Let us consider one CSV (comma separated values) file created for our practice session which are mentioned below and present on the current directory. First row is the name of the fields and remaining lines are actual data.

  File Name: empmst.csv.

  Empno,Empname,DOB,City,Salary
  20101,Rishav Sen,13-05-1969,Kolkata,50000
  23819,Biplab Mitra,17-03-1991,Kolkata,52000
  67120,Sanjay Roy,14-10-1998,Delhi,45000
  78123,Aniket Sinha,01-05-1982,Chennai,65000
  37621,Ajay Sharma,12-08-2010,Mumbai,45000
  55123,Dipak Arora,31-09-1990,Delhi,43000

  Now we create the Pandas DataFrame (DF) from the above mentioned CSV file. To use 'csv' format file **we need to call the method 'read_csv' from Pandas library**.
  Note: Column names are case sensitive.

- **Read a CSV file**

```
# Using 'read_csv' to call one CSV file from the disk
empdf=pd.read_csv("empmst.csv")
print(empdf)      # Records with indexes = 0,1,2,....and Empno is fileld 0, Empname is field 1 so on...
```

- **Make Empno as the index**

```
# Now we make Empno as the index
empdf = pd.read_csv('empmst.csv', index_col='Empno')
print(empdf)              # Outputs records with 'Empno' as index column
```

- **To get the information of a DF, use info() method**

```
print(empdf.info())
```

- **Column list, no. of records, shape, name of index column**

```
print(empdf.columns)        # Print list of columns / fields
print(len(empdf))           # Print No of records
print(empdf.index.name)     # Name of the Index column, Empno
```

- **Usage of Integer location (iloc) in DF e.g. → empdf.iloc[[row selection], [column selection]]**

```
print(empdf.iloc[[1,2]])          # Select row 1,2
print(empdf.iloc[[0,3]])          # Select row 0,3
# Use slice operator to select a range
print(empdf.iloc[0:3])            # Select row 0,1,2 but not 3. Please note this is one pair of [ ]

# Select rows and columns
print(empdf.iloc[0:3, [1,2]])             # Select rows 0,1,2 and columns 1,2. (city and salary)
                                          # Empname is col 0, DOB is col 1, City is col 2 so on

print(empdf.iloc[[1,4], [0,2]])           # Select row 1,4 with columns 0,2
```

```
print(empdf.iloc[:,1])          # Select column 1 (i.e. DOB) for all rows
print(empdf.iloc[:, -1])        # Last column (Salary) of all rows
```

- **Usage of 'loc' in DF**

```
print(empdf.loc[:, 'Empname'])                          # Print Empname for all rows
print(empdf.loc[67120, 'Empname'])                      # Selects Sanjay Roy of Empno=67120
print(empdf.loc[67120, ['Empname', Salary']])           # Print Empname and Salary of  Empno=67120
print(empdf.loc[[20101, 67120], ['Empname', 'Salary']]) # Empname & Salary for Empno=20101, 67120
```

- **Access columns by names**

```
print(empdf['Empname'])
print(empdf[['Empname', 'Salary']])
```

- **Access columns by Attributes**

```
print(empdf.Empname, empdf.Salary)
```

- **Selecting rows of a DF by Boolean conditions**

```
print(empdf.Salary > 50000)     # Prints Empno/ or index col and 'True or False' depending on the
                                # condition.
print(empdf[empdf.Salary > 50000])     # Prints rows with all columns where condition is True
# Multiple condition
print(empdf[(empdf.Salary < 60000) & (empdf.Salary >=50000)])
```

- **Making copy of a DF by 'copy' method**

```
empcopy = empdf.copy()      # Made a copy of the original DF
empcopy= empdf[:2].copy()   # Copies 0,1 row from empdf to empcopy
```

- **<u>Modifying the structure and contents of a DF</u>**

  - Rows and columns can be added & removed and data can be modified
  - Columns and index labels can be renamed

- **Renaming columns**

```
# This rename command changed the original DF with new column name
empdf=empdf.rename(columns={'Salary':'EmpSalary'})          # Salary changed to EmpSalary
print(empdf)            # Displays with new column name
```

- **Adding/inserting columns to a DF**

```
# Add a new column called 'HRA' (as 30% of Salary) at the end.
empdf['HRA'] = empdf.EmpSalary * 0.3
print(empdf)            # Shows all columns plus the new column HRA as 30% of Salary

# Insert a column at a position using 'insert()'
empdf.insert(4, 'DA', empdf.EmpSalary * 0.25)    # Col DA added at position 4
print(empdf)                                     # Prints the updated DF
```

- **<u>Updating contents of a Column in DF</u>**

```
# Update the column DA with new percentage 10%
empdf['DA'] = empdf.EmpSalary * 0.10
print(empdf)
```

```
# To update City(Column 2)for Row 3 via 'iloc'
empdf.iloc[3, [2]] = 'Kochi'
print(empdf)

# To update Salary of Empno=67120 using 'loc'
# Note : Change of 'EmpSalary' for Empno=67120 does not update  HRA or DA
# which are dependent on EmpSalary
empdf.loc[67120, 'EmpSalary'] = 25000
```

- **Deleting Columns in a DF**

  Columns can be deleted from a DF by 'del'
  'del' → **will delete the column from the DF**

```
# Drop the column 'DA' affecting the original DF
del empdf['DA']          # Delete the column DA
print(empdf)             # Prints the DF without DA.
```

- **Deleting Rows in a DF**

```
# Delete rows using 'drop'
empdf.drop([67120, 55123])     # Displays without these two rows but original DF is unchanged
# To delete and update the DF, use 'inplace=True' as parameter
empdf.drop([67120, 55123], inplace=True)
# Remove by index positions
empdf.drop(empdf.index[0:3], inplace=True)    # drop rows 0,1,2 and update the original DF
print(empdf)                    # Prints rows with first three rows removed
```

- **<u>Adding Rows & Columns in a DF via Enlargement</u>**

  Parameter for '.loc' specifies the index label where rows are to be placed.
  If the label does not exist → values are appended at the given index label
  If the Index label exists → values in the specified rows are updated.

```
# Adds a record in the DF with Empno, Empname, DOB, City, Salary
empdf.loc[90990] = ['Mainak Gupta', '16-11-1998', 'Chennai', 45000]    # Changes original DF
print(empdf)                    # Output shows the newly added record in the DF
# To remove from the original DF, use 'inplace=True'
empdf.drop(empdf.index[0:3], inplace=True)
print(empdf)             # Prints rows with first three rows removed
```

- **Accessing a sub frame in a DF**

```
# Selecting rows & columns
subdf = empdf[1:4][['Empname', 'DOB']]    # Rows 1,2,3 with columns 'Empname' and 'DOB'
print(subdf)
```

- **<u>Some useful tips while accessing CSV file from Python Pandas library</u>**

  - **Skip column header details from csv file. Mention your own columns names via 'names'**

```
# Change column names while reading CSV file
```

```
import pandas as pd
empdf1 = pd.read_csv('empmst.csv', header=0, names=['Emp ID', 'Name', 'Birth Dt', 'City', 'Basic'],
                                                    index_col=['Emp ID'])
print(empdf1)
```

- **To load specific columns from the '.csv' file using 'usecols'**

```
# Read selective columns like Empno, Empname and DOB from the .csv file with 'Empno' as index
empdf2 = pd.read_csv('empmst.csv', usecols=['Empno', 'Empname', 'DOB'], index_col=['Empno'])
print(empdf2)              # Displays only two columns with 'Empno' as index
```

- **Saving DF to a CSV file**

We can write the DF data to a CSV file by the method 'to_csv()'

```
# Read the data into a DF from a CSV file
empdf5  = pd.read_csv('empmst.csv', index_col='Empno')
# Add a column HRA as 25% of Salary
empdf5['HRA']= empdf5.Salary * 0.25
# Add a column Total as Salary + HRA
empdf5['Total']=empdf5.Salary + empdf5.HRA
# Now save this DF onto a disk file 'empmst_new.csv' in .csv format
empdf5.to_csv('empmst_new.csv', index_label='Empno')   # This new file would now contain HRA, Total
```

Now check the file called 'empmst_new.csv' created in your system by 'notepad' or 'MS Word'

- **General Field Delimited Data**

General field delimited data where separator can be '|', '%'..... etc. Comma is a special case of generic delimiter which are managed by 'csv' file. One can mention the separator via 'sep' keyword and using the method called 'read_table'. Like 'to_csv', Pandas do not have any 'to_table' method, instead it uses 'to_csv'. Instead of 'read_table' one can use 'read_csv' with proper delimiter mentioned with 'sep' parameter.

```
# Read a file called 'studmst.txt' where the separator is '|' (pipe character) with 'Roll' as index
# m1, m2, m3 are three marks
studdf=pd.read_table('studmst.txt', sep='|', index_col='Roll')
print(studdf)              # Print Roll as index with m1, m2, m3 as columns

# We can write our old DF called 'empdf' to a disk file with a different separator
empdf.to_csv('empmst1.txt', sep='|')     # Check the disk file to find the data separated by '|'
```

- **Removing noise rows from a field delimited data file, crucial step towards Data Science and Machine Learning**

Noise rows can be

1. Unwanted characters in headers/footers
2. Blank lines

Use 'skiprows' to remove noise and use 'skip_footer' to remove noise at the end. While using 'skip_footer', mention 'engine='python'"

```
#To skip 1, 2 line from the beginning
e6 = pd.read_csv('empmst.csv', index_col='Empno', skiprows=[1,2])
#  To skip line 1,2 from beginning and 1 row from footer
```

```
e7 = pd.read_csv('empmst.csv', index_col='Empno', skiprows=[1,2], skipfooter=1,engine='python')

# Reading only first 3 rows use 'nrows'
e8 = pd.read_csv('empmst.csv', index_col='Empno', nrows=3)
```