PANDAS BASICS

In [1]:

```
#We import as follows
import numpy as np
```

In [2]:

```
import pandas as pd
```

In [3]:

```
#Basic Data Structure in PANDAS :
#Series (One-Dimensional Array) ex : int,str and py obj
#DataFrame (Two-Dimensional Array) ex : rows and columns ie tables.
```

Object Creation.

In [6]:

```
#Creating a series by passing a list of values,letting pandas create a default RangeIndex.
s = pd.Series ([1,3,5,np.nan,6,8])
```

In [7]:

```
s
```

Out[7]:

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

In [9]:

```
#Creating a DataFrame by passing a NumPy array with a datetime index using data_range() and labeled columns :
dates = pd.date_range("20130101", periods = 6)
```

In [10]:

```
dates
```

Out[10]:

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [11]:

```
df = pd.DataFrame(np.random.randn(6,4), index = dates, columns = list ("ABCD"))
```

In [12]:

```
df
```

Out[12]:

	A	B	C	D
2013-01-01	-0.212211	-0.496392	0.341294	-1.674359
2013-01-02	2.618365	1.144702	-0.675272	-1.827035
2013-01-03	-1.134542	0.527121	-0.036757	1.481542
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667
2013-01-05	0.173427	-1.569299	-0.998047	-0.115180
2013-01-06	0.479126	-0.570549	1.427960	1.007732

In [16]:

```
#Creating a DataFrame by passing a dictionary of objects where the keys are the column labels and the values ar
df2 = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20130102"),
        "C": pd.Series(1, index = list (range(4)), dtype = "float32"),
        "D": np.array([3] * 4, dtype = "int32"),
        "E": pd.Categorical(["test", "train", "test", "train"]),
        "F": "foo",
    }
)
```

In [17]:

```
df2
```

Out[17]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

In [18]:

```
df2.dtypes
```

Out[18]:

```
A           float64
B     datetime64[s]
C           float32
D             int32
E          category
F            object
dtype: object
```

Viewing Data.

In [22]:

```
#Use DataFrame.head() and DataFrame.tail() to view the Top and Bottom rows of the frame respvtly.
df.head()
```

Out[22]:

	A	B	C	D
2013-01-01	-0.212211	-0.496392	0.341294	-1.674359
2013-01-02	2.618365	1.144702	-0.675272	-1.827035
2013-01-03	-1.134542	0.527121	-0.036757	1.481542
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667
2013-01-05	0.173427	-1.569299	-0.998047	-0.115180

In [24]:

```
df.tail(3)
```

Out[24]:

	A	B	C	D
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667
2013-01-05	0.173427	-1.569299	-0.998047	-0.115180
2013-01-06	0.479126	-0.570549	1.427960	1.007732

In [25]:

```
#Display the DataFrame.index or DataFrame.columns :
df.index
```

Out[25]:

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [26]:

```
df.columns
```

Out[26]:

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

In [27]:

```
#Returning a NumPy representation of the data with DataFrame.to_numpy() without index/column labels.
df.to_numpy
```

Out[27]:

```
<bound method DataFrame.to_numpy of                    A         B         C         D
2013-01-01 -0.212211 -0.496392  0.341294 -1.674359
2013-01-02  2.618365  1.144702 -0.675272 -1.827035
2013-01-03 -1.134542  0.527121 -0.036757  1.481542
2013-01-04 -0.477778 -0.690549 -1.445837  0.244667
2013-01-05  0.173427 -1.569299 -0.998047 -0.115180
2013-01-06  0.479126 -0.570549  1.427960  1.007732>
```

In [28]:

```
#NumPy arrays have one dtype for the entire array while PANDAS DataFrames have one dtype per column.
#When you call DataFrame.to_Numpy(),PANDAS will find the NumPy dtype that can hold all of the dtypes in the Data
#If the common dtype is "object",DataFrame.to_numpy() will require copyong data.
df2.dtypes
```

Out[28]:

```
A           float64
B     datetime64[s]
C           float32
D             int32
E          category
F            object
dtype: object
```

In [29]:

```
df2.to_numpy()
```

Out[29]:

```
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']],
      dtype=object)
```

In [30]:

```
# describe() shows a quick statistic summary of your data.
df.describe()
```

Out[30]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.241064	-0.275828	-0.231110	-0.147106
std	1.290527	0.963660	1.037204	1.363474
min	-1.134542	-1.569299	-1.445837	-1.827035
25%	-0.411386	-0.660549	-0.917353	-1.284564
50%	-0.019392	-0.533470	-0.356014	0.064743
75%	0.402701	0.271243	0.246781	0.816966
max	2.618365	1.144702	1.427960	1.481542

In [31]:

```
#Transposing T of your Data.
df.T
```

Out[31]:

	2013-01-01	2013-01-02	2013-01-03	2013-01-04	2013-01-05	2013-01-06
A	-0.212211	2.618365	-1.134542	-0.477778	0.173427	0.479126
B	-0.496392	1.144702	0.527121	-0.690549	-1.569299	-0.570549
C	0.341294	-0.675272	-0.036757	-1.445837	-0.998047	1.427960
D	-1.674359	-1.827035	1.481542	0.244667	-0.115180	1.007732

In [34]:

```
#DataFrame.sort_index() sorts by an axis :
df.sort_index(axis = 1, ascending = False)
```

Out[34]:

	D	C	B	A
2013-01-01	-1.674359	0.341294	-0.496392	-0.212211
2013-01-02	-1.827035	-0.675272	1.144702	2.618365
2013-01-03	1.481542	-0.036757	0.527121	-1.134542
2013-01-04	0.244667	-1.445837	-0.690549	-0.477778
2013-01-05	-0.115180	-0.998047	-1.569299	0.173427
2013-01-06	1.007732	1.427960	-0.570549	0.479126

In [35]:

```
#DataFrame.sort_values() sorts by values :
df.sort_values(by = "B")
```

Out[35]:

	A	B	C	D
2013-01-05	0.173427	-1.569299	-0.998047	-0.115180
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667
2013-01-06	0.479126	-0.570549	1.427960	1.007732
2013-01-01	-0.212211	-0.496392	0.341294	-1.674359
2013-01-03	-1.134542	0.527121	-0.036757	1.481542
2013-01-02	2.618365	1.144702	-0.675272	-1.827035

Selection.

In [36]:

```
# Optimized PANDAS data acces methods :
# df.at() , df.iat(), df.loc(), df.iloc().
```

In [37]:

```
# Get ([])
#For a DataFrame,passing a single label selects a columns and yields a series = df.a :
df.A
```

Out[37]:

```
2013-01-01   -0.212211
2013-01-02    2.618365
2013-01-03   -1.134542
2013-01-04   -0.477778
2013-01-05    0.173427
2013-01-06    0.479126
Freq: D, Name: A, dtype: float64
```

In [39]:

```
# passing a slice : , selects matching rows:
df[0:3]
```

Out[39]:

	A	B	C	D
2013-01-01	-0.212211	-0.496392	0.341294	-1.674359
2013-01-02	2.618365	1.144702	-0.675272	-1.827035
2013-01-03	-1.134542	0.527121	-0.036757	1.481542

In [40]:

```
df["20130102":"20130104"]
```

Out[40]:

	A	B	C	D
2013-01-02	2.618365	1.144702	-0.675272	-1.827035
2013-01-03	-1.134542	0.527121	-0.036757	1.481542
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667

In [41]:

```
#Selection by Label -
df.loc[dates[0]]
```

Out[41]:

```
A   -0.212211
B   -0.496392
C    0.341294
D   -1.674359
Name: 2013-01-01 00:00:00, dtype: float64
```

In [42]:

```
#Selecting all rows (:) with a select column labels :
df.loc[:, ["A","B"]]
```

Out[42]:

	A	B
2013-01-01	-0.212211	-0.496392
2013-01-02	2.618365	1.144702
2013-01-03	-1.134542	0.527121
2013-01-04	-0.477778	-0.690549
2013-01-05	0.173427	-1.569299
2013-01-06	0.479126	-0.570549

In [45]:

```
#For label slicing , both end points are included:
df.loc["20130102":"20130104", ["C","D"]]
```

Out[45]:

	C	D
2013-01-02	-0.675272	-1.827035
2013-01-03	-0.036757	1.481542
2013-01-04	-1.445837	0.244667

In [46]:

```
#Selecting a single row and column returns Scalar :
df.loc[dates[0], "A"]
```

Out[46]:

```
-0.2122114741798462
```

In [47]:

```
#for getting fast access to a scalar:
df.at[dates[0], "A"]
```

Out[47]:

```
-0.2122114741798462
```

In [48]:

```
#Selection by position
# df.iloc() and df.iat()
```

In [49]:

```
df.iloc[3]
```

Out[49]:

```
A   -0.477778
B   -0.690549
C   -1.445837
D    0.244667
Name: 2013-01-04 00:00:00, dtype: float64
```

In [52]:

```
df.iloc[3:5, 0:2]
```

Out[52]:

	A	B
2013-01-04	-0.477778	-0.690549
2013-01-05	0.173427	-1.569299

In [53]:

```
df.iloc[[1,2,4], [0,2]]
```

Out[53]:

	A	C
2013-01-02	2.618365	-0.675272
2013-01-03	-1.134542	-0.036757
2013-01-05	0.173427	-0.998047

In [54]:

```
df.iloc[1:3, :]
```

Out[54]:

	A	B	C	D
2013-01-02	2.618365	1.144702	-0.675272	-1.827035
2013-01-03	-1.134542	0.527121	-0.036757	1.481542

In [55]:

```
df.iloc[0:3, :1]
```

Out[55]:

	A
2013-01-01	-0.212211
2013-01-02	2.618365
2013-01-03	-1.134542

In [56]:

```
#For getting value explicitly :
df.iloc[1]
```

Out[56]:

```
A    2.618365
B    1.144702
C   -0.675272
D   -1.827035
Name: 2013-01-02 00:00:00, dtype: float64
```

In [57]:

```
df.iloc[1:1]
```

Out[57]:

	A	B	C	D

In [58]:

```
df.iloc[1, 1]
```

Out[58]:

```
1.1447022749803721
```

In [60]:

```
# Boolean Indexing.
#Select rows where df.A is greater than 0.
df[df["A"] > 2]
```

Out[60]:

	A	B	C	D
2013-01-02	2.618365	1.144702	-0.675272	-1.827035

In [61]:

```
#Selecting values from a Dataframe whre boolean condition is met.
df[df > 0]
```

Out[61]:

	A	B	C	D
2013-01-01	NaN	NaN	0.341294	NaN
2013-01-02	2.618365	1.144702	NaN	NaN
2013-01-03	NaN	0.527121	NaN	1.481542
2013-01-04	NaN	NaN	NaN	0.244667
2013-01-05	0.173427	NaN	NaN	NaN
2013-01-06	0.479126	NaN	1.427960	1.007732

In [64]:

```
#Using isin() method for filterng:
df2 = df.copy()
```

In [65]:

```
df2["E"] = ["one", "one", "two", "three", "four", "three"]
```

In [66]:

```
df2
```

Out[66]:

	A	B	C	D	E
2013-01-01	-0.212211	-0.496392	0.341294	-1.674359	one
2013-01-02	2.618365	1.144702	-0.675272	-1.827035	one
2013-01-03	-1.134542	0.527121	-0.036757	1.481542	two
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667	three
2013-01-05	0.173427	-1.569299	-0.998047	-0.115180	four
2013-01-06	0.479126	-0.570549	1.427960	1.007732	three

In [67]:

```
df2[df2["E"].isin(["two", "four"])]
```

Out[67]:

	A	B	C	D	E
2013-01-03	-1.134542	0.527121	-0.036757	1.481542	two
2013-01-05	0.173427	-1.569299	-0.998047	-0.115180	four

In [68]:

```
df2[df2["E"].isin(["one", "three"])]
```

Out[68]:

	A	B	C	D	E
2013-01-01	-0.212211	-0.496392	0.341294	-1.674359	one
2013-01-02	2.618365	1.144702	-0.675272	-1.827035	one
2013-01-04	-0.477778	-0.690549	-1.445837	0.244667	three
2013-01-06	0.479126	-0.570549	1.427960	1.007732	three

Setting.

In [69]:

```
#Setting a new column automatically aligns the data by the indexes:
s1 = pd.Series([1,2,3,4,5,6], index = pd.date_range("20130102", periods = 6))
```

In [70]:

```
s1
```

Out[70]:

```
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64
```

In [71]:

```
df["F"] = s1
```

In [72]:

```
#Setting values by label
df.at[dates[0], "A"] = 0
```

In [74]:

```
#Setting values by position:
df.iat[0, 1] = 0
```

In [76]:

```
#Setting by assigning with a NumPy array :
df.loc[:, "D"] = np.array([5] * len(df))
```

In [77]:

```
df
```

Out[77]:

	A	B	C	D	F
2013-01-01	0.000000	0.000000	0.341294	5.0	NaN
2013-01-02	2.618365	1.144702	-0.675272	5.0	1.0
2013-01-03	-1.134542	0.527121	-0.036757	5.0	2.0
2013-01-04	-0.477778	-0.690549	-1.445837	5.0	3.0
2013-01-05	0.173427	-1.569299	-0.998047	5.0	4.0
2013-01-06	0.479126	-0.570549	1.427960	5.0	5.0

Missing Data.

In [79]:

```
#For NumPy dtypes, np.nan represents missing data.
#It is not included in computations.
```

In [80]:

```
#Re-indexing allows you to change/add/delete the index on a specified axis.
df1 = df.reindex(index = dates[0:4], columns = list(df.columns) + ["E"])
```

In [81]:

```
df1
```

Out[81]:

	A	B	C	D	F	E
2013-01-01	0.000000	0.000000	0.341294	5.0	NaN	NaN
2013-01-02	2.618365	1.144702	-0.675272	5.0	1.0	NaN
2013-01-03	-1.134542	0.527121	-0.036757	5.0	2.0	NaN
2013-01-04	-0.477778	-0.690549	-1.445837	5.0	3.0	NaN

In [84]:

```
#df.dropna() drops any rows that have missing data:
df1.dropna(how = "any")
```

Out[84]:

	A	B	C	D	F	E

In [85]:

```
#df.fillna() fills the missing data :
df1.fillna(value = 5)
```

Out[85]:

	A	B	C	D	F	E
2013-01-01	0.000000	0.000000	0.341294	5.0	5.0	5.0
2013-01-02	2.618365	1.144702	-0.675272	5.0	1.0	5.0
2013-01-03	-1.134542	0.527121	-0.036757	5.0	2.0	5.0
2013-01-04	-0.477778	-0.690549	-1.445837	5.0	3.0	5.0

In [87]:

```
#isna() gets the boolean mask  where values are nan :
pd.isna(df1)
```

Out[87]:

	A	B	C	D	F	E
2013-01-01	False	False	False	False	True	True
2013-01-02	False	False	False	False	False	True
2013-01-03	False	False	False	False	False	True
2013-01-04	False	False	False	False	False	True

Operations.

In [90]:

```
#STATS-
#Operations in general exclude missing data.
# Note : axis 0 = column , axis 1 = row.
```

In [91]:

```
#Calculation of mean value for each column:
df.mean(axis=0)
```

Out[91]:

```
A    0.276433
B   -0.193095
C   -0.231110
D    5.000000
F    3.000000
dtype: float64
```

In [92]:

```
#Calculation of mean value for each row :
df.mean(axis = 1)
```

Out[92]:

```
2013-01-01    1.335323
2013-01-02    1.817559
2013-01-03    1.271164
2013-01-04    1.077167
2013-01-05    1.321216
2013-01-06    2.267307
Freq: D, dtype: float64
```

In [93]:

```
# USER DEFINED FUNCTIONS - lamda X [ df.agg() and df.tranform() ]
# The above applies a user defined function that reduces or broadcasts its result respectively.
df.agg(lambda x: np.mean(x) * 5.6)
```

```
Out[93]: A     1.548025
         B    -1.081335
         C    -1.294215
         D    28.000000
         F    16.800000
         dtype: float64
```

```
In [94]: df.transform(lambda x: x * 101.2)
```

Out[94]:

	A	B	C	D	F
2013-01-01	0.000000	0.000000	34.538947	506.0	NaN
2013-01-02	264.978572	115.843870	-68.337506	506.0	101.2
2013-01-03	-114.815644	53.344634	-3.719816	506.0	202.4
2013-01-04	-48.351087	-69.883531	-146.318676	506.0	303.6
2013-01-05	17.550785	-158.813023	-101.002384	506.0	404.8
2013-01-06	48.487536	-57.739512	144.509522	506.0	506.0

```
In [96]: # VALUE COUNTS -
         s = pd.Series(np.random.randint(0,7, size = 10))
```

```
In [97]: s
```

```
Out[97]: 0    0
         1    0
         2    2
         3    2
         4    4
         5    2
         6    0
         7    4
         8    1
         9    0
         dtype: int32
```

```
In [99]: s.value_counts()
```

```
Out[99]: 0    4
         2    3
         4    2
         1    1
         Name: count, dtype: int64
```

```
In [100… # STRING METHODS -
         # Series is equipped with a set of string processing methods in the str attribute that make it easy to operate
         s = pd.Series(["A","B","C","D",np.nan,"PraneshH"])
```

```
In [101… s.str.lower()
```

```
Out[101… 0          a
         1          b
         2          c
         3          d
         4        NaN
         5    pranesh
         dtype: object
```

Merge.

```
In [104… # Concatenation
         # Concatenating PANDAS objects together row-wise with concat().
```

```
In [105… df = pd.DataFrame(np.random.randn(10, 4))
```

```
In [106… df
```

Out[106...

	0	1	2	3
0	0.390479	1.484150	-0.271168	0.427987
1	0.855291	1.042513	-0.265487	-1.078957
2	1.027565	-0.397128	0.039541	-1.026591
3	1.126499	-0.014406	-0.278593	-1.036125
4	0.895349	0.491619	-0.323037	0.369785
5	-0.497965	-1.106655	0.689489	-1.128373
6	0.228841	-1.265396	1.042296	-1.028925
7	-0.377584	-1.003859	-0.255499	-1.076286
8	-0.998901	-0.234052	-1.090360	-1.055999
9	-1.081615	1.080591	-0.148728	-1.367942

In [107...

```
#break it into pieces
pieces = [df[:3],df[3:7],df[7:]]
```

In [108...

```
pieces
```

Out[108...

```
[          0         1         2         3
 0  0.390479  1.484150 -0.271168  0.427987
 1  0.855291  1.042513 -0.265487 -1.078957
 2  1.027565 -0.397128  0.039541 -1.026591,
           0         1         2         3
 3  1.126499 -0.014406 -0.278593 -1.036125
 4  0.895349  0.491619 -0.323037  0.369785
 5 -0.497965 -1.106655  0.689489 -1.128373
 6  0.228841 -1.265396  1.042296 -1.028925,
           0         1         2         3
 7 -0.377584 -1.003859 -0.255499 -1.076286
 8 -0.998901 -0.234052 -1.090360 -1.055999
 9 -1.081615  1.080591 -0.148728 -1.367942]
```

In [109...

```
pd.concat(pieces)
```

Out[109...

	0	1	2	3
0	0.390479	1.484150	-0.271168	0.427987
1	0.855291	1.042513	-0.265487	-1.078957
2	1.027565	-0.397128	0.039541	-1.026591
3	1.126499	-0.014406	-0.278593	-1.036125
4	0.895349	0.491619	-0.323037	0.369785
5	-0.497965	-1.106655	0.689489	-1.128373
6	0.228841	-1.265396	1.042296	-1.028925
7	-0.377584	-1.003859	-0.255499	-1.076286
8	-0.998901	-0.234052	-1.090360	-1.055999
9	-1.081615	1.080591	-0.148728	-1.367942

In [110...

```
# JOIN -
# merge() enables SQL style join types along specific columns.
```

In [111...

```
left = pd.DataFrame({"key": ["abc","def"], "lval": [1,2]})
```

In [112...

```
right = pd.DataFrame({"key": ["abc","def"], "lval": [4,5]})
```

In [113...

```
left
```

Out[113...

	key	lval
0	abc	1
1	def	2

In [114...

```
right
```

Out[114…

	key	lval
0	abc	4
1	def	5

In [117…

```
pd.merge(left,right, on="key")
```

Out[117…

	key	lval_x	lval_y
0	abc	1	4
1	def	2	5

Grouping.

In [118…

```
# By "Group By" we are reffering to a process involving one or more of the foll steps:
# Spilliting the data into groups based on some criteria
# Applying a function to each group independently
# Combining the resuts into a data structure
```

In [121…

```
df = pd.DataFrame(
    {
    "A": np.random.randn(8),
    "B": np.random.randn(8),
    "C": np.random.randn(8),
    "D": np.random.randn(8),
    }
)
```

In [122…

```
df
```

Out[122…

	A	B	C	D
0	-1.018267	0.159905	1.181607	-0.828122
1	-0.857054	-0.294918	1.021169	1.787522
2	-0.094020	-0.164254	-0.652354	-0.151381
3	1.041970	-0.097208	1.279389	0.552806
4	0.427791	1.342883	-0.082630	1.321911
5	-1.115415	-0.214185	0.123916	0.383595
6	-1.163154	1.409206	-0.336527	-0.729700
7	-2.036835	0.481497	1.196360	1.260712

In [123…

```
# Applying df.groupby.sum() :
```

In [124…

```
df.groupby("A")[["C","D"]].sum()
```

Out[124…

	C	D
A		
-2.036835	1.196360	1.260712
-1.163154	-0.336527	-0.729700
-1.115415	0.123916	0.383595
-1.018267	1.181607	-0.828122
-0.857054	1.021169	1.787522
-0.094020	-0.652354	-0.151381
0.427791	-0.082630	1.321911
1.041970	1.279389	0.552806

Reshapping.

In [125…

```
# STACK -
arrays = [
     ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ["one", "two", "one", "two", "one", "two", "one", "two"],
]
```

In [126…

```
index = pd.MultiIndex.from_arrays(arrays, names=["first", "second"])
```

In [127…

```
arrays
```

Out[127…

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

In [128…

```
df2 = df[:4]
```

In [129…

```
df2
```

Out[129…

	A	B	C	D
0	-1.018267	0.159905	1.181607	-0.828122
1	-0.857054	-0.294918	1.021169	1.787522
2	-0.094020	-0.164254	-0.652354	-0.151381
3	1.041970	-0.097208	1.279389	0.552806

In [130…

```
# The stack() method “compresses” a level in the DataFrame’s columns:
```

In [132…

```
stacked = df2.stack(future_stack = True)
```

In [133…

```
stacked
```

Out[133…

```
0  A   -1.018267
   B    0.159905
   C    1.181607
   D   -0.828122
1  A   -0.857054
   B   -0.294918
   C    1.021169
   D    1.787522
2  A   -0.094020
   B   -0.164254
   C   -0.652354
   D   -0.151381
3  A    1.041970
   B   -0.097208
   C    1.279389
   D    0.552806
dtype: float64
```

In [134…

```
# the inverse operation of stack() is unstack(), which by default unstacks the last level:
```

In [135…

```
stacked.unstack(0)
```

Out[135…

	0	1	2	3
A	-1.018267	-0.857054	-0.094020	1.041970
B	0.159905	-0.294918	-0.164254	-0.097208
C	1.181607	1.021169	-0.652354	1.279389
D	-0.828122	1.787522	-0.151381	0.552806

In [136…

```
# PIVOT TABLES
# pivot_table() pivots a DataFrame specifying the values, index and columns
```

In [137…

```
pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
```

Out[137…

	C	**-0.652354**	**-0.336527**	**-0.082630**	**0.123916**	**1.021169**	**1.181607**	**1.196360**	**1.279389**
A	**B**								
-2.036835	**0.481497**	NaN	NaN	NaN	NaN	NaN	NaN	1.260712	NaN
-1.163154	**1.409206**	NaN	-0.7297	NaN	NaN	NaN	NaN	NaN	NaN
-1.115415	**-0.214185**	NaN	NaN	NaN	0.383595	NaN	NaN	NaN	NaN
-1.018267	**0.159905**	NaN	NaN	NaN	NaN	NaN	-0.828122	NaN	NaN
-0.857054	**-0.294918**	NaN	NaN	NaN	NaN	1.787522	NaN	NaN	NaN
-0.094020	**-0.164254**	-0.151381	NaN	NaN	NaN	NaN	NaN	NaN	NaN
0.427791	**1.342883**	NaN	NaN	1.321911	NaN	NaN	NaN	NaN	NaN
1.041970	**-0.097208**	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.552806

Time Series

In [155…

```
# Converting secondly data into 5-minutely data
rng = pd.date_range("1/1/2012", periods=10, freq="s")
```

In [156…

```
rng
```

Out[156…

```
DatetimeIndex(['2012-01-01 00:00:00', '2012-01-01 00:00:01',
               '2012-01-01 00:00:02', '2012-01-01 00:00:03',
               '2012-01-01 00:00:04', '2012-01-01 00:00:05',
               '2012-01-01 00:00:06', '2012-01-01 00:00:07',
               '2012-01-01 00:00:08', '2012-01-01 00:00:09'],
              dtype='datetime64[ns]', freq='S')
```

In [157…

```
ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
```

In [158…

```
ts
```

Out[158…

```
2012-01-01 00:00:00     95
2012-01-01 00:00:01    365
2012-01-01 00:00:02    309
2012-01-01 00:00:03    474
2012-01-01 00:00:04    436
2012-01-01 00:00:05    380
2012-01-01 00:00:06     87
2012-01-01 00:00:07    220
2012-01-01 00:00:08     24
2012-01-01 00:00:09    146
Freq: S, dtype: int32
```

In [159…

```
ts.resample("5min").sum()
```

Out[159…

```
2012-01-01    2536
Freq: 5T, dtype: int32
```

Importing and Exporting Data.

In [160…

```
# CSV file
# While writing to csv file : using df.to_csv()
# While Reading from a csv file : using read_csv()
```

In [161…

```
df = pd.DataFrame(np.random.randint(0, 5, (10, 5)))
```

In [163…

```
df.to_csv("Pranesh.csv")
```

In [164…

```
pd.read_csv("Pranesh.csv")
```

Out[164…

	Unnamed: 0	0	1	2	3	4
0	0	1	4	1	2	4
1	1	2	1	3	3	2
2	2	4	4	0	0	0
3	3	0	3	0	1	4
4	4	3	3	4	3	2
5	5	3	3	2	1	3
6	6	0	1	3	2	4
7	7	4	0	2	2	0
8	8	1	3	4	0	4
9	9	1	1	4	4	1

In [165…

```
# Parquet file
df.to_parquet("Pranesh.parquet")
```

In [166…

```
pd.read_parquet("Pranesh.parquet")
```

Out[166…

	0	1	2	3	4
0	1	4	1	2	4
1	2	1	3	3	2
2	4	4	0	0	0
3	0	3	0	1	4
4	3	3	4	3	2
5	3	3	2	1	3
6	0	1	3	2	4
7	4	0	2	2	0
8	1	3	4	0	4
9	1	1	4	4	1

In [167…

```
# Excel file
# Writing to an excel file using df.to_excel()
# Reading from an excel file using read_excel()
```

In [168…

```
df.to_excel("Pranesh.xlsx", sheet_name="Sheet1")
```

In [169…

```
pd.read_excel("Pranesh.xlsx", "Sheet1", index_col=None, na_values=["NA"])
```

Out[169…

	Unnamed: 0	0	1	2	3	4
0	0	1	4	1	2	4
1	1	2	1	3	3	2
2	2	4	4	0	0	0
3	3	0	3	0	1	4
4	4	3	3	4	3	2
5	5	3	3	2	1	3
6	6	0	1	3	2	4
7	7	4	0	2	2	0
8	8	1	3	4	0	4
9	9	1	1	4	4	1

- THE END -

@Pranesh notes.

In [ ]: