UNIT-3 Part-2
Data manipulation with Pandas

Syllabus: Data manipulation with Pandas – data indexing and selection – operating on data – missing data – hierarchical indexing – combining datasets –aggregation and grouping – pivot tables.

Pandas

→ Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame.

→ DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.

→ As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

Pandas Objects

(Fundamental Pandas Data Structures)

→ Three fundamental Pandas data structures are:

- Series
- DataFrame
- Index.

The Pandas Series Object

→ A Pandas Series is a one-dimensional array of indexed data.

→ Example:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
```

Output:

```
0    0.25
1    0.50
3    0.75
3    1.00
```

→ The Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes. The index is an array-like object of type pd.Index,

Example:

```
print(data.values)
print(data.index)
```

Output:

```
[0.25 0.5  0.75 1.  ]
RangeIndex(start=0, stop=4, step=1)
```

→ The essential difference between NumPy one-dimensional array and pandas Series is the presence of the index: while the NumPy array has an *implicitly defined* integer index used to access the values, the Pandas Series has an *explicitly defined* index associated with the values.

→ This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type.

→ Example:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],index=['a', 'b', 'c', 'd'])
print(data)
```

Output:

```
a    0.25
b    0.50
c    0.75
d    1.00
```

→ We can even use non-contiguous or non-sequential indices:

Example:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
print(data)
```

Output:

```
2    0.25
5    0.50
3    0.75
7    1.00
```

Constructing Series objects

→ The general syntax to create pandas Series object is

```
pd.Series(data, index=index)
```

where index is an optional argument, and data can be one of many entities.

- data can be a list or NumPy array, in which case index defaults to an integer sequence
- data can be a scalar, which is repeated to fill the specified index
- data can be a dictionary, in which index defaults to the sorted dictionary keys

→ Example program:

```
import pandas as pd
import numpy as np
arr=np.arange(10,60,10)
li=[10,20,30,40,50]
s=10
dic={'1st':10,'2nd':20,'3rd':30,'4th':40,'5th':50}
ser1 = pd.Series(arr)   #A one-dimensional ndarray
```

```
ser2 = pd.Series(li)      # A Python list
ser3 = pd.Series(s)      #A scalar value
ser4 =pd.Series(s,index=['a','b','c','d','e'])
ser5 = pd.Series(dic)  #A Python dictionary
print(ser1)
print(ser2)
print(ser3)
print(ser4)
print(ser5)
```

Output:

```
0    10
1    20
2    30
3    40
4    50

0    10
1    20
2    30
3    40
4    50

0    10

a    10
b    10
c    10
d    10
e    10

1st    10
2nd    20
3rd    30
4th    40
5th    50
```

<u>The Pandas DataFrame Object</u>

- → The DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.
- → A DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.

→ We can think of a DataFrame as a sequence of aligned (they share the same index) Series objects.

→ Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

→ Example:

```
import pandas as pd
df=pd.DataFrame([[10,20],[30,40],[50,60]])
print(df)
df=pd.DataFrame([[10,20],[30,40],[50,60]],columns=['col1', 'col2'])
print(df)
df=pd.DataFrame([[10,20],[30,40],[50,60]],index=['row1', 'row2', 'row3'])
df=pd.DataFrame([[10,20],[30,40],[50,60]],columns=['col1', 'col2'],
                index=['row1', 'row2', 'row3'])
print(df)
```

Output:

```
    0   1
0  10  20
1  30  40
2  50  60

   col1  col2
0    10    20
1    30    40
2    50    60

      col1  col2
row1    10    20
row2    30    40
row3    50    60
```

Constructing DataFrame objects

→ A Pandas DataFrame can be constructed in a variety of ways.

- From a single Series object
- From List of Dicts
- From a dictionary of Series objects
- From a two-dimensional NumPy array
- From a NumPy structured array

From a single Series object:

→ A DataFrame is a collection of Series objects, and a single column DataFrame can be constructed from a single Series:

Example:

```
import pandas as pd
```

```
marksList = {'kumar':89,'Rao':78,'Ali':67,'Singh':96}
marks = pd.Series(markslist)
df= pd.DataFrame(marks,columns=['Marks'])
print(df)
Output:
         Marks
kumar    89
Rao      78
Ali      67
Singh    96
```

From List of Dicts:

→ Any list of dictionaries can be made into a DataFrame.

→ Example:

```
import pandas as pd
import numpy as np
data = [{'a':i,'b':2*i} for i in range(3)]
print(pd.DataFrame(data))
#alternate way of defining
l1={'a':0,'b':0}
l2={'a':1,'b':2}
l3={'a':2,'b':4}
data = [l1,l2,l3]
print('\n',pd.DataFrame(data))
```

Output:

```
   a  b
0  0  0
1  1  2
2  2  4
   a  b
0  0  0
1  1  2
2  2  4
```

From a dictionary of Series objects:

→ A DataFrame can be constructed from a dictionary of Series objects

→ Example:

```
import pandas as pd
markslist = {'kumar':89,'Rao':78,'Ali':67,'Singh':96}
ageslist = {'kumar':21,'Rao':22,'Ali':19,'Singh':20}
marks = pd.Series(markslist)
ages = pd.Series(ageslist)
df = pd.DataFrame({'marks': marks,'ages': ages})
print(df)
```

Output:
```
          marks  ages
kumar     89    21
Rao       78    22
Ali       67    19
Singh     96    20
```

From a two-dimensional NumPy array.

→ Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each

→ **Example:**
```
import pandas as pd
import numpy as np
df=pd.DataFrame(np.arange(1,7,1).reshape(3,2),
columns=['col1', 'col2'],
index=['row1', 'row2', 'row3'])
print(df)
```

Output:
```
        col1  col2
row1     1     2
row2     3     4
row3     5     6
```

From a NumPy structured array.

→ A Pandas DataFrame operates much like a structured array, and can be created directly from one:

Example:
```
import numpy as np
import pandas as pd
sa = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
print(pd.DataFrame(sa))
```
Output:
```
   A  B
0 0 0.0
1 0 0.0
2 0 0.0
```

Pandas Index Object

→ Both the Series and DataFrame objects contain an explicit index using which we reference and modify data.

→ This Index object is an interesting structure in itself, and it can be thought of either as an immutable array or as an ordered set.

Example:

```
import pandas as pd
rind = pd.Index(['row1','row2','row3','row4'])
cind =pd.Index(['col1'])
ser = pd.Series([100,200,300,400],index=rind)
df = pd.DataFrame(ser,columns=cind)
print(df)
```

Output:

```
      col1
row1   100
row2   200
row3   300
row4   400
```

```
import pandas as pd
rind = pd.Index(['row1','row2','row3','row4'])
ser1 = pd.Series([10,20,30,40],index=rind)
ser2 = pd.Series([50,60,70,80],index=rind)
frame={'col1':ser1,'col2':ser2}
df = pd.DataFrame(frame)
print(df)
```

Output:

```
      col1  col2
row1    10    50
row2    20    60
row3    30    70
row4    40    80
```

Operating on Data in Pandas

→ Pandas inherit much of this functionality from NumPy, and the ufuncs. So Pandas having the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.).

→ For unary operations like negation and trigonometric functions, these ufuncs will preserve index and column labels in the output.

→ For binary operations such as addition and multiplication, Pandas will automatically align indices when passing the objects to the ufunc.

→ The universal functions are working in series and DataFrames by

- Index preservation

- Index alignment

Index Preservation

→ Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects.

→ We can use all arithmetic and special universal functions as in NumPy on pandas. In outputs the index will preserved (maintained) as shown below.

```
import pandas as pd
import numpy as np
ser = pd.Series([10,20,30,40])
df = pd.DataFrame(np.arange(1,13,1).reshape(3,4),columns=['A', 'B', 'C',
'D'])
print(df)
print(np.add(ser,5))  # the indices preserved for series
print(np.add(df,10))  # the indices preserved for DataFrame
```

Index Alignment in series

→ Pandas will align indices in the process of performing the operation. This is very convenient when we are working with incomplete data, as we'll.

→ suppose we are combining two different data sources, then the index will aligned accordingly.

→ Exampe:

```
import numpy as np
import pandas as pd
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
print(A + B)
print(A.add(B))    #equivalent to A+B
print(A.add(B,fill_value=0)) #fill value for any elements in A or B that
might be missing
```

Index Alignment in DataFrame

A similar type of alignment takes place for both columns and indices when we are performing operations on DataFrames.

Example:

```
import numpy as np
import pandas as pd
A = pd.DataFrame(np.arange(1,5,1).reshape(2,2),columns =list('AB'))
B = pd.DataFrame(np.arange(1,10,1).reshape(3,3),columns =list('BAC'))
print(A)
print(B)
print(A+B)
print(A.add(B,fill_value=0))
fill = A.stack().mean()
print(A.add(B,fill_value=fill))
```

Output:

```
   A  B
0  1  2
1  3  4
   B  ...  C
0  1  ...  3
1  4  ...  6
2  7  ...  9

[3 rows x 3 columns]
     A  ...   C
0  3.0  ... NaN
1  8.0  ... NaN
2  NaN  ... NaN

[3 rows x 3 columns]
     A  ...    C
0  3.0  ...  3.0
1  8.0  ...  6.0
2  8.0  ...  9.0

[3 rows x 3 columns]
      A  ...     C
0   3.0  ...   5.5
1   8.0  ...   8.5
2  10.5  ...  11.5

[3 rows x 3 columns]
```

Operations between DataFrame and Series

→ When we are performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained.

→ Operations between a DataFrame and a Series are similar to operations between a two-dimensional and one-dimensional NumPy array.

Example:

```
import numpy as np
import pandas as pd
ser = pd.Series([10,20])
df = pd.DataFrame([[100,200],[300,400]])
print(ser)
print(df)
print(df.subtract(ser))
print(df.subtract(ser,axis=0))
```

Output:

```
0    10
1    20

     0    1
0  100  200
1  300  400
     0    1
0   90  180
1  290  380
     0    1
0   90  190
1  280  380
```

Data Selection in DataFrame

DataFrame as a dictionary

Example1:

```
import pandas as pd
ser1 = pd.Series([10,20,30,40],index = ['row1','row2','row3','row4'])
ser2 = pd.Series([50,60,70,80],index = ['row1','row2','row3','row4'])
data = pd.DataFrame({'col1':ser1,'col2':ser2})
print(data)
print(data['col1']) # dict style
print(data.col1)    # attribute style
data['sum'] = data['col1']+data['col2']
print(data)
```

Output:

```
      col1  col2
row1    10    50
row2    20    60
row3    30    70
row4    40    80
row1    10
row2    20
row3    30
row4    40

row1    10
row2    20
row3    30
row4    40
```

```
      col1  ...  sum
row1    10  ...   60
row2    20  ...   80
row3    30  ...  100
row4    40  ...  120

[4 rows x 3 columns]
```

Example2:

```
import pandas as pd
markslist = {'kumar':89,'Rao':78,'Ali':67,'Singh':96}
ageslist = {'kumar':21,'Rao':22,'Ali':19,'Singh':20}
marks = pd.Series(markslist)
ages = pd.Series(ageslist)
data = pd.DataFrame({'marks': marks,'ages': ages})
print(data)
print(data['marks'])
print(data.marks)
data['ratio'] = data['marks'] / data['ages']
print(data)
```

Output:

```
       marks  ages
kumar     89    21
Rao       78    22
Ali       67    19
Singh     96    20

kumar    89
Rao      78
Ali      67
Singh    96

kumar    89
Rao      78
Ali      67
Singh    96

       marks  ...     ratio
kumar     89  ...  4.238095
```

```
Rao       78  ...  3.545455
Ali       67  ...  3.526316
Singh     96  ...  4.800000
[4 rows x 3 columns]
```

DataFrame as two-dimensional array

Example1:

```
import pandas as pd
ser1 = pd.Series([10,20,30,40],index = ['row1','row2','row3','row4'])
ser2 = pd.Series([50,60,70,80],index = ['row1','row2','row3','row4'])
data = pd.DataFrame({'col1':ser1,'col2':ser2})
print(data)
print(data.values)
print(data.T)
print(data.value[0])
print(data.iloc[:3,:1])
print(data.loc[:'row3',:'col1'])
#print(data.ix[:3,:'col1'])
```

Output:

```
      col1  col2
row1    10    50
row2    20    60
row3    30    70
row4    40    80
[[10 50]
 [20 60]
 [30 70]
 [40 80]]
      row1  ...  row4
col1    10  ...    40
col2    50  ...    80
 [2 rows x 4 columns]
 [10 50]

      col1
row1    10
row2    20
row3    30
      col1
row1    10
row2    20
row3    30
```

Example2:

```
import pandas as pd
markslist = {'kumar':89,'Rao':78,'Ali':67,'Singh':96}
ageslist = {'kumar':21,'Rao':22,'Ali':19,'Singh':20}
marks = pd.Series(markslist)
ages = pd.Series(ageslist)
data = pd.DataFrame({'marks': marks,'ages': ages})
print(data)
print(data.values)
print(data.T)
```

Output:

```
        marks  ages
kumar     89    21
Rao       78    22
Ali       67    19
Singh     96    20
[[89 21]
 [78 22]
 [67 19]
 [96 20]]
        kumar  ...  Singh
marks     89  ...     96
ages      21  ...     20
[2 rows x 4 columns]
```

Handling Missing Data

→ A number of schemes have been developed to indicate the presence of missing data in a table or DataFrame.

→ Generally, they revolve around one of two strategies: using a **mask** that globally indicates missing values, or choosing a **sentinel value** that indicates a missing entry.

→ In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

→ In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

→ Example:

```
import numpy as np
import pandas as pd
arr1 =np.array([1,2,3,4])
print(arr1)
```

```
print(arr1.sum())
arr2 =np.array([1,None,3,4])
print(arr2)
#print(arr2.sum())
arr3 =np.array([1,np.nan,3,4])
print(arr3)
print(arr3.sum())
print(np.nansum(arr3))
```

Output:

```
[1 2 3 4]
10
[1 None 3 4]
[ 1. nan  3.  4.]
nan
8.0
```

Missing Data in Pandas

→ The way in which Pandas handles missing values is constrained by its NumPy package, which does not have a built-in notion of NA values for non floating- point data types.

→ NumPy supports fourteen basic integer types once we account for available precisions, signedness, and endianness of the encoding.

→ Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package.

→ Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floatingpoint NaN value, and the Python None object.

→ This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

None: Pythonic missing data

→ The first sentinel value used by Pandas is None, a Python singleton object that is often used for missing data in Python code. Because None is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects)

→ This dtype=object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects.

NaN: Missing numerical data

→ NaN is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation.

NaN and None in Pandas

→ NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably.

Example:

```
import numpy as np
import pandas as pd
ser = pd.Series([1,np.nan,2,None])
print(ser)
df = pd.DataFrame([[1,None],[3,np.nan],[None,6],[np.nan,8]])
print(df)
```

Output:

```
0    1.0
1    NaN
2    2.0
3    NaN

     0    1
0  1.0  NaN
1  3.0  NaN
2  NaN  6.0
3  NaN  8.0
```

Operating on Null Values

→ There are several useful methods for detecting, removing, and replacing null values in Pandas data structures.

→ They are:

- isnull() - Generate a Boolean mask indicating missing values
- notnull() - Opposite of isnull()
- dropna() - Return a filtered version of the data
- fillna() - Return a copy of the data with missing values filled or imputed

Detecting null values

Pandas data structures have two useful methods for detecting null data: isnull() and notnull().

Example:

```
import numpy as np
import pandas as pd
ser = pd.Series([1,np.nan,'hello',None])
df = pd.DataFrame([[np.nan,10,'hai'],[20,30,'wow']])
print(ser)
print(ser.isnull())
print(ser.notnull())
```

```
print(df)
print(df.isnull())
print(df.notnull())

0       1
1      NaN
2      hello
3      None

0    False
1     True
2    False
3     True

0     True
1     False
2     True
3     False

  0  ...    2
0   NaN  ...  hai
1  20.0  ...  wow

[2 rows x 3 columns]

      0  ...      2
0   True  ...  False
1  False  ...  False

[2 rows x 3 columns]

       0  ...     2
0  False  ...  True
1   True  ...  True

[2 rows x 3 columns]
```

Dropping Null values

```
import numpy as np
import pandas as pd
ser = pd.Series([1,np.nan,'hello',None])
df = pd.DataFrame([[np.nan,10,'hai'],[20,30,'wow']])
```

```
print(ser)
print(df)
print(ser.dropna())
print(df.dropna())
print(df.dropna(axis =1))
print(df.dropna(axis ='columns'))  #equivalent to axis =1
```

```
0       1
1     NaN
2   hello
3    None

     0  ...    2
0   NaN  ...  hai
1  20.0  ...  wow

[2 rows x 3 columns]

0       1
2   hello

     0  ...    2
1  20.0  ...  wow

[1 rows x 3 columns]

 1    2
0  10  hai
1  30  wow

 1    2
0  10  hai
1  30  wow
```

Example:

```
import numpy as np
import pandas as pd
df = pd.DataFrame([[np.nan,10,'hai',None],[20,30,'wow',None]])
print(df)
print(df.dropna())
print(df.dropna(axis =1))
print(df.dropna(axis ='columns')) #equivalent to axis =1
```

```
print(df.dropna(axis ='columns',how='all'))
print(df.dropna(axis ='columns',thresh=2))
```

Output:

```
     0  ...     3
0   NaN  ...  None
1  20.0  ...  None

[2 rows x 4 columns]

Empty DataFrame
Columns: [0, 1, 2, 3]
Index: []

    1    2
0  10  hai
1  30  wow

    1    2
0  10  hai
1  30  wow

     0  ...    2
0   NaN  ...  hai
1  20.0  ...  wow

[2 rows x 3 columns]

    1    2
0  10  hai
1  30  wow
```

Filling null values in DataFrame

```
import numpy as np
import pandas as pd
ser = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
print(ser)
print(ser.fillna(0))
print(ser.fillna(method='ffill'))
print(ser.fillna(method='bfill'))
```

Output:

```
a    1.0
```

```
b    NaN
c    2.0
d    NaN
e    3.0

a    1.0
b    0.0
c    2.0
d    0.0
e    3.0

a    1.0
b    1.0
c    2.0
d    2.0
e    3.0

a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
```

Filling null values in DataFrame

Example

```
import numpy as np
import pandas as pd
df = pd.DataFrame([[1, np.nan, 2,None],
                   [2, 3, 5, None],
                   [np.nan, 4, 6, None]])
print(df)
print(df.fillna(method='ffill', axis=1))
print(df.fillna(method='bfill', axis=1))
print(df.fillna(method='ffill', axis=0))
print(df.fillna(method='bfill', axis=0))
```

Output:

```
     0  ...     3
0  1.0  ...  None
1  2.0  ...  None
2  NaN  ...  None

     0  ...    3
```

```
0  1.0  ...  2.0
1  2.0  ...  5.0
2  NaN  ...  6.0

     0  ...   3
0  1.0  ... NaN
1  2.0  ... NaN
2  4.0  ... NaN

]
     0  ...     3
0  1.0  ...  None
1  2.0  ...  None
2  2.0  ...  None

     0  ...     3
0  1.0  ...  None
1  2.0  ...  None
2  NaN  ...  None
```

Hierarchical Indexing

- → *Hierarchical indexing* (also known as *multi-indexing*) is used to incorporate multiple index *levels* within a single index.
- → In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.
- → A Multiply Indexed Series: Here we represent two-dimensional data within a one-dimensional Series.

  Example:

```
import numpy as np
import pandas as pd
ser = pd.Series([10,20,30,40,50,60],index = [[1,1,1,2,2,2,],
        ['a','b','c','a','b','c']])
print(ser)
ser.index.names = ['ind1','ind2']
print(ser)
```

  Output:

```
1  a    10
   b    20
   c    30
2  a    40
   b    50
```

```
c    60

ind1  ind2
1     a       10
      b       20
      c       30
2     a       40
      b       50
      c       60
```

→ A Multiply Indexed DataFrame:

Example:

```
import numpy as np
import pandas as pd
data = [[25,24],[28,26],[29,28],[27,26],[30,29],[28,27]]
ind = [['1201','1201','1264','1264','12C7','12C7'],
          ['mid1','mid2','mid1','mid2','mid1','mid2']]
col = ['DS','DO']
df = pd.DataFrame(data,index=ind,columns=col)
print(df)
df.index.names =['rollNo','mid']
print(df)
```

Output:

```
              DS  DO
1201 mid1  25  24
       mid2  28  26
1264 mid1  29  28
       mid2  27  26
12C7 mid1  30  29
       mid2  28  27

              DS  DO
rollNo  mid
1201   mid1  25  24
       mid2  28  26
1264   mid1  29  28
       mid2  27  26
12C7   mid1  30  29
       mid2  28  27
```

Example:

Python program to create following table of data

Dept   Other

```
               DS  DO  MOB EPC
1201 mid1   25  24    23     15
     mid2   28  26    23     21
1264 mid1   29  28    27     26
     mid2   27  26    24     25
12C7 mid1    30  29    28     27
     mid2    28  27    25    26
```

Program:

```
import numpy as np
import pandas as pd
data = [[25,24,23,15],[28,26,23,21],[29,28,27,26],[27,26,24,25],[30,29,28,27],
        [28,27,25,26]]
ind = [['1201','1201','1264','1264','12C7','12C7'],
       ['mid1','mid2','mid1','mid2','mid1','mid2']]
col = [['Dept','Dept','Other','Other'],['DS','DO','MOB','EPC']]
df = pd.DataFrame(data,index=ind,columns=col)
print(df.to_string())
```

Output:

```
                 Dept     Other
               DS  DO  MOB EPC
1201 mid1   25  24    23     15
     mid2   28  26    23     21
1264 mid1   29  28    27     26
     mid2   27  26    24     25
12C7 mid1    30  29    28     27
     mid2    28  27    25    26
```

Example:

Python program to create following table:

```
  Type             Dept     Other
Sub            DS  DO   MOB EPC
RollNo Mid
1201   mid1   25  24    23  15
       mid2   28  26    23  21
1264   mid1   29  28    27  26
       mid2   27  26    24  25
12C7   mid1  30  29    28  27
       mid2   28  27    25  26
```

Program:

```
import numpy as np
import pandas as pd
data = [[25,24,23,15],[28,26,23,21],[29,28,27,26],[27,26,24,25],[30,29,28,27],
```

```
          [28,27,25,26]]
ind = [['1201','1201','1264','1264','12C7','12C7'],
         ['mid1','mid2','mid1','mid2','mid1','mid2']]
col = [['Dept','Dept','Other','Other'],['DS','DO','MOB','EPC']]
df = pd.DataFrame(data,index=ind,columns=col)
df.index.names =['RollNo','Mid']
df.columns.names =['Type','Sub']
print(df.to_string())
```

Output:

```
Type             Dept    Other
Sub             DS  DO   MOB EPC
RollNo Mid
1201   mid1   25  24    23  15
       mid2   28  26    23  21
1264   mid1   29  28    27  26
       mid2   27  26    24  25
12C7   mid1   30  29    28  27
       mid2   28  27    25  26
```

Combining Datasets

→ Some of the most interesting studies of data come from combining different data sources.

→ These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the dataset.

→ These operations can be:
- simple concatenation of Series and DataFrames with the pd.concat function
- in-memory merges and joins implemented in Pandas.

Simple Concatenation with pd.concat

→ Pandas has a function, pd.concat(), which has a similar syntax to np.concatenate but contains a number of other options

→ pd.concat() can be used for a simple concatenation of Series or DataFrame objects, just as np.concatenate() can be used for simple concatenations of arrays

```
import pandas as pd
import numpy as np
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
print(pd.concat([ser1, ser2]))
```

Output:

```
1 A
2 B
3 C
4 D
5 E
6 F
```

→ **Concatenation in data frame:**

```
import pandas as pd
import numpy as np
df1 =pd.DataFrame([[10,20],[30,40]],index=[1,2],columns=['A','B'])
df2 =pd.DataFrame([[50,60],[70,80]],index=[1,2],columns=['A','B'])
print(df1); print(df2); print(pd.concat([df1, df2]))
```

Output:

```
 A   B
1  10  20
2  30  40

 A   B
1  50  60
2  70  80

 A   B
1  10  20
2  30  40
1  50  60
2  70  80
```

→ By default, the concatenation takes place row-wise within the DataFrame (i.e., axis=0). Like np.concatenate, pd.concat allows specification of an axis along which concatenation will take place.

Example:

```
import pandas as pd
import numpy as np
df1 =pd.DataFrame([[10,20],[30,40]],index=[1,2],columns=['A','B'])
df2 =pd.DataFrame([[50,60],[70,80]],index=[1,2],columns=['A','B'])
print(df1); print(df2);
print(pd.concat([df1, df2],axis=1).to_string())
```

Output:

```
    A   B
1  10  20
2  30  40
    C   D
1  50  60
```

```
2  70  80
   A   B   C   D
1  10  20  50  60
2  30  40  70  80
```

→ By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the join and join_axes parameters of the concatenate function. By default, the join is a union of the input columns (join='outer'), but we can change this to an intersection of the columns using join='inner':

Example:

```
import pandas as pd
import numpy as np
df1 =pd.DataFrame([[1,2,3],[4,5,6]],index=[1,2],columns=['A','B','C'])
df2
=pd.DataFrame([[7,8,9],[10,11,12]],index=[1,2],columns=['B','C','D'])
print(df1.to_string()); print(df2.to_string())
print(pd.concat([df1, df2]).to_string())
print(pd.concat([df1, df2],join='inner'))
```

Output:

```
  A  B  C
1  1  2  3
2  4  5  6

   B   C   D
1   7   8   9
2  10  11  12

     A   B   C     D
1  1.0   2   3   NaN
2  4.0   5   6   NaN
1  NaN   7   8   9.0
2  NaN  10  11  12.0

    B   C
1   2   3
2   5   6
1   7   8
2  10  11
```

The append() method

→ Series and DataFrame objects have an append method that can accomplish the concatenation in fewer keystrokes.
→ For example, rather than calling pd.concat([df1, df2]), we can simply call df1.append(df2):
print(df1); **print**(df2); **print**(df1.append(df2))

Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations.

Categories of Joins

- One-to-one joins
- Many-to-one joins
- Many-to-many joins

One – to – one joins

The simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation.

df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
'hire_date': [2004, 2008, 2012, 2014]})
print(df1); **print**(df2)

employee group employee hire_date
0 Bob Accounting 0 Lisa 2004
1 Jake Engineering 1 Bob 2008
2 Lisa Engineering 2 Jake 2012
3 Sue HR 3 Sue 2014

To combine this information into a single DataFrame, we can use the pd.merge() function

df3 = pd.merge(df1, df2)
print(df3)

employee group hire_date
0 Bob Accounting 2008
1 Jake Engineering 2012
2 Lisa Engineering 2004
3 Sue HR 2014

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate.

df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
'supervisor': ['Carly', 'Guido', 'Steve']})
pd.merge(df3, df4)

employee group hire_date supervisor
0 Bob Accounting 2008 Carly
1 Jake Engineering 2012 Guido
2 Lisa Engineering 2004 Guido
3 Sue HR 2014 Steve

The resulting DataFrame has an additional column with the “supervisor” information, where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example.

df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR', 'HR'], 'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organization']})
pd.merge(df1, df5)

employee group skills
0 Bob Accounting math
1 Bob Accounting spreadsheets
2 Jake Engineering coding
3 Jake Engineering linux
4 Lisa Engineering coding
5 Lisa Engineering linux
6 Sue HR spreadsheets
7 Sue HR organization

Aggregation and Grouping

→ An essential piece of analysis of large data is efficient summarization: computing aggregations like sum(), mean(), median(), min(), and max(), in which a single number gives insight into the nature of a potentially large dataset.

→ Aggregation in pandas can be performed by:
  - Simple Aggregation
  - Operations based on the concept of a groupby.

Simple Aggregation in Pandas

→ As with a one dimensional NumPy array, for a Pandas Series the aggregates return a single value:
Example:
```
import pandas as pd
import numpy as np
ser = pd.Series([10,20,30,40,50])
print(ser.sum())
```

```
print(ser.mean())
Output:
150
30.0
```

→ For a DataFrame, by default the aggregates return results within each column. By specifying the axis argument, we can instead aggregate within each row.

Example:

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'A':np.arange(1,6),
                   'B':np.arange(10,60,10)})
print(df.sum())
print(df.mean())
print(df.sum(axis ='columns'))
print(df.mean(axis = 'columns'))
```

Output:

```
A     15
B    150
dtype: int64
A     3.0
B    30.0
dtype: float64
0    11
1    22
2    33
3    44
4    55
dtype: int64
0     5.5
1    11.0
2    16.5
3    22.0
4    27.5
dtype: float64
```

→ Pandas Series and DataFrames include all of the common aggregates .In addition, there is a convenience method describe() that computes several common aggregates for each column and returns the result.

Example:

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'A':np.arange(1,6),
                   'B':np.arange(10,60,10)})
```

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

Output:

```
             A          B
count  5.000000   5.000000
mean   3.000000  30.000000
std    1.581139  15.811388
min    1.000000  10.000000
25%    2.000000  20.000000
50%    3.000000  30.000000
75%    4.000000  40.000000
max    5.000000  50.000000
```

→ Some of other built-in Pandas aggregations are:

Table 3-3. Listing of Pandas aggregation methods

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

GroupBy: Split, Apply, Combine

- → The groupby operation llows to quickly and efficiently compute aggregates on subsets of data.
- → The groupby operation is used to aggregate conditionally on some label or index.
- → The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.
  - The ***split*** step involves breaking up and grouping a DataFrame depending on the value of the specified key.
  - The ***apply*** step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
  - The ***combine*** step merges the results of these operations into an output array.
- → Example program

```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame({'key':['A','B','C','A','B','C'],
                   'data':np.arange(1,7)},columns=['key','data'])
print(df)
print(df.groupby('key').sum())
Output:
  key  data
0   A     1
1   B     2
2   C     3
3   A     4
4   B     5
5   C     6

     data
key
A       5
B       7
C       9
```

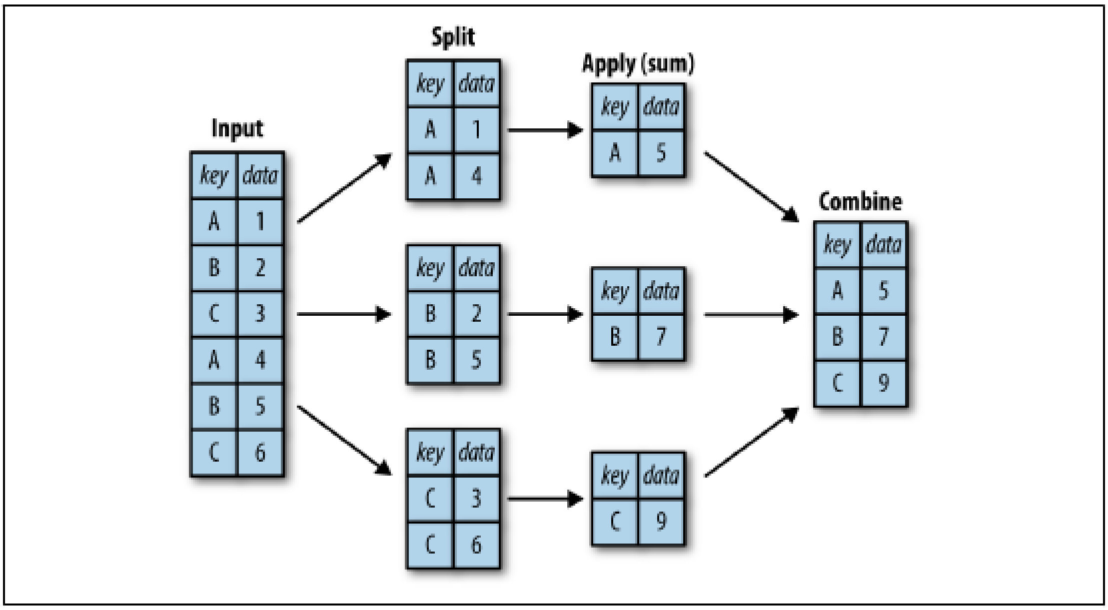

Figure 3-1. A visual representation of a groupby operation

Pivot Tables

→ A *pivot table* is a similar to GroupBy operation that is commonly seen in spreadsheets and other programs that operate on tabular data.

→ The pivot table takes simple column wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data.

→ We can think of pivot tables as essentially a *multidimensional* version of GroupBy aggregation. i.e., we can split-apply- combine, but both the split and the combine happen across not a one dimensional index, but across a two-dimensional grid.

→ **Pivot Table Syntax:** The full call signature of the pivot_table method of DataFrames is as follows:

```
DataFrame.pivot_table(data, values=None, index=None,
                      columns=None,aggfunc='mean',
                      fill_value=None, margins=False,
                      dropna=True, margins_name='All')
```

where

data : pandas dataframe
index : feature that allows to group data
values : feature to aggregates on
columns: displays the values horizontally on top of the resultant table
fill_value and dropna, have to do with missing data
The aggfunc keyword controls what type of aggregation is applied, which is a mean by default.
margins_name: compute totals along each grouping.

→ **Example:**

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'Name':['Kumar','Rao','Ali','Singh'],
            'Job':['FullTimeEmployee','Intern','PartTime
Employee','FullTimeEmployee'],
            'Dept':['Admin','Tech','Admin','management'],
            'YOJ':[2018,2019,2018,2010],
            'Sal':[20000,50000,10000,20000]})
print(df.to_string())
output = pd.pivot_table(data=df,index=['Job'],columns = ['Dept'],
                        values ='Sal',aggfunc ='mean')
print('\n')
print(output.to_string())
```

Output:

```
    Name                Job        Dept   YOJ    Sal
0  Kumar    FullTimeEmployee       Admin  2018  20000
1    Rao              Intern        Tech  2019  50000
2    Ali   PartTime Employee       Admin  2018  10000
3  Singh    FullTimeEmployee  management  2010  20000


Dept                  Admin      Tech  management
Job
FullTimeEmployee    20000.0       NaN     20000.0
Intern                  NaN   50000.0         NaN
PartTime Employee   10000.0       NaN         NaN
```

Tutorial Questions:

1. Explain the fundamental data objects with its construction in pandas
2. Briefly explain the hierarchical indexing with examples
3. What is pivot table? Explain it clearly
4. Demonstrate data indexing and selection in Pandas Series and DataFrame objects.
5. Write short note on Operating on Data in Pandas
6. Demonstrate different methods of constructing MultiIndex.
7. How to handle missing data in pandas
8. Illustrate different approaches to combine data from multiple sources in pandas
9. Explore aggregation and grouping in Pandas
10. Briefly explore and demonstrate different methods for Operating on Null Values

Assignment Questions:

1. Write a python program to illustrate different ways of creating pandas Series
2. Write a python program to illustrate different ways of creating pandas DataFrame
3. Write a python program to illustrate detecting null values in pandas dataFrame
4. Write a python program to illustrate dropping null values in pandas DataFrame
5. Write a python program to illustrate filling null values in pandas DataFrame
6. Write a python program to illustrate creating different ways of pandas MutiIndex
7. Write a python program to illustrate indexing, slicing, Boolean indexing and fancy indexing in MultiIndex.
8. Write a python program to illustrate merging two data sets with joins(inner, left and right) in pandas
9. Write a python program to illustrate **GroupBy** operation of pandas.
10. Write a python program to illustrate **pivot table** in pandas.