Data Wrangling

Join, Combine and Reshape

In [1]:

```
# Sometimes it is very difficult anlyze the data that is not arraged  or stored properl
y
```

Importance of Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that enables you to have multiple (two or more) index levels on an axix

In [2]:

```
# This help us to work with higher dimensional data in lower dimensional form
```

In [4]:

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

In [ ]:

```

```

In [5]:

```
# Let's create a Series with list of lists as its index
data_hi = pd.Series(np.random.randn(9),
          index=[['A', 'A', 'A', 'B', 'B', 'C', 'C', 'D', 'D'],
                 [1, 2, 3, 1, 4, 1, 2, 2, 4]])
data_hi
# You can easily observe that the Series has a MultiIndex as its index
```

Out[5]:

```
A  1    0.351449
   2    1.278593
   3    1.159270
B  1   -0.733105
   4   -0.265540
C  1   -0.204688
   2    0.778994
D  2    0.210338
   4   -0.145815
dtype: float64
```

In [27]:

```
# The index (axis labels) of the Series.
pd.Series.index?
```

In [6]:

```
# Let's access the index to what type of Index it is using 'Series.index' method
data_hi.index
```

Out[6]:

```
MultiIndex(levels=[['A', 'B', 'C', 'D'], [1, 2, 3, 4]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 3, 0, 1, 1,
3]])
```

In [7]:

```
# Partial indexing is possible with the hierachically indexed object
data_hi['A']
```

Out[7]:

```
1    0.351449
2    1.278593
3    1.159270
dtype: float64
```

In [8]:

```
# slicing is also possible
data_hi['A':'C']
```

Out[8]:

```
A  1    0.351449
   2    1.278593
   3    1.159270
B  1   -0.733105
   4   -0.265540
C  1   -0.204688
   2    0.778994
dtype: float64
```

In [10]:

```
# selecting a particular index is also possible: pass a list of index to be selected in
side the indexing object i.e. data_hi[index]
data_hi[['A', 'C']]
```

Out[10]:

```
A  1    0.351449
   2    1.278593
   3    1.159270
C  1   -0.204688
   2    0.778994
dtype: float64
```

In [11]:

```
# we can also use explicit loc to index the Series object
data_hi.loc[:, 1]
```

Out[11]:

```
A    0.351449
B   -0.733105
C   -0.204688
dtype: float64
```

In [12]:

```
# we can rearrange the data into a DataFrame object using 'unstack' method
# Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame.
# The level involved will automatically get sorted.
pd.Series.unstack?
```

In [13]:

```
# The missing values are replaced with NaN by default. However we can replace the missi
ng values using 'fill_value=' argument
data_hi.unstack()
```

Out[13]:

	1	2	3	4
A	0.351449	1.278593	1.15927	NaN
B	-0.733105	NaN	NaN	-0.265540
C	-0.204688	0.778994	NaN	NaN
D	NaN	0.210338	NaN	-0.145815

In [14]:

```
data_hi.unstack(fill_value=0)
```

Out[14]:

	1	2	3	4
A	0.351449	1.278593	1.15927	0.000000
B	-0.733105	0.000000	0.00000	-0.265540
C	-0.204688	0.778994	0.00000	0.000000
D	0.000000	0.210338	0.00000	-0.145815

In [15]:

```
# The inverse operation of 'unstack' is 'stack'
# you can check doc_string of it using 'pd.Series.unstack.stack?'
data_hi.unstack().stack()
```

Out[15]:

```
A  1    0.351449
   2    1.278593
   3    1.159270
B  1   -0.733105
   4   -0.265540
C  1   -0.204688
   2    0.778994
D  2    0.210338
   4   -0.145815
dtype: float64
```

In [17]:

```
# A DataFrame in otherwords can have MultiIndex in either index axis (row index or colu
mn index)
# In DataFrame the name 'index' infers to row index by default, but while selecting the
column index is used implicitly
df_hi = pd.DataFrame(np.arange(12).reshape((4, 3)),
                     index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                     columns=[['one', 'one', 'three'],
                              ['Green', 'Red', 'Green']])
df_hi
```

Out[17]:

		one		three
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [57]:

```
pd.names?
```

```
Object `pd.names` not found.
```

In [68]:

```
# We can check the documentation of any attribute using 'help' command also.
#help(pd.MultiIndex)
#pd.MultiIndex?
```

In [21]:

```
# we can easily rename the row index and column index names with the help of 'DataFram
e.index.names' n 'ataFrame.column.names' attributes
# change the row index lavel names
df_hi.index.names = ['val1', 'val2']
```

In [23]:

```
# change the column index level names
df_hi.columns.names = ['number', 'color']
```

In [24]:

```
# Let's display the DF
df_hi
```

Out[24]:

	number	one		three
	color	Green	Red	Green
val1	val2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [26]:

```
# now we can apply partial indexing on DF
df_hi['one']
```

Out[26]:

	color	Green	Red
val1	val2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

In [69]:

```
# WE can create our own MultiIndex and can be reused whenever required using 'MultiInde
x' method
```

How Reordering and Sorting of Index Levels Takes Place?

In [70]:

```
# we can use 'swaplevel' method: it will rearrange the order of the levels on an axis o
r sort the data by the values
# in one specific level
# Docstring: Swap levels i and j in a MultiIndex on a particular axis

pd.DataFrame.swaplevel?
```

In [71]:

```
# let's swap the row index levels first. it is the default operation
df_hi.swaplevel('val1', 'val2', axis=0)
```

Out[71]:

	number	one		three
	color	Green	Red	Green
val2	val1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

In [72]:

```
# We can swap the column index levels by passing 'axis=1'
df_hi.swaplevel('number', 'color', axis=1)
```

Out[72]:

	color	Green	Red	Green
	number	one	one	three
val1	val2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [77]:

```
# sort_index, on the other hand, sorts the data using only the values in a single level
# Docstring: Sort object by labels (along an axis)
pd.DataFrame.sort_index?
```

In [75]:

```
# this mehod sorts the index level lexographically by the indicated level
# let's sort the index labels with level=1, i.e on second index level of Row's MultiInd
ex
df_hi.sort_index(level=1)
```

Out[75]:

	number	one		three
	color	Green	Red	Green
val1	val2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

In [78]:

```
# # let's sort the index labels with level=0, i.e on first index level of Row's MultiIn
dex
df_hi.sort_index(level=0)
```

Out[78]:

	number	one		three
	color	Green	Red	Green
val1	val2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [79]:

```
# we can apply sort_index on top of swaplevel to get the sorted index of the swaped lev
els
df_hi.swaplevel(0, 1).sort_index(level=0)
# Now the val2 becomes level=0 and val1 becomes level=1 and then sorted on val2, i.e on
val2
```

Out[79]:

	number	one		three
	color	Green	Red	Green
val2	val1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

How To Get The Summary Statistics By Level?

In [87]:

```
# Docstring: Return the sum of the values for the requested axis
pd.DataFrame.sum?
```

In [82]:

```
# The Pandas Series and DataFrame have a level option for their descriptive and summary
stastistics
print(df_hi)
df_hi.sum(level='val1') # all 'a' and 'b's are grouped first and then sum is applied
```

```
number      one     three
color     Green Red Green
val1 val2
a    1        0   1     2
     2        3   4     5
b    1        6   7     8
     2        9  10    11
```

Out[82]:

number	one		three
color	Green	Red	Green
val1			
a	3	5	7
b	15	17	19

In [83]:

```
print(df_hi)
df_hi.sum(level='val2') # all '1' and '2's are grouped first and then sum is applied
```

```
number      one     three
color     Green Red Green
val1 val2
a    1        0   1     2
     2        3   4     5
b    1        6   7     8
     2        9  10    11
```

Out[83]:

number	one		three
color	Green	Red	Green
val2			
1	6	8	10
2	12	14	16

In [85]:

```
# we can do across the columns using 'axis=1' and then the column name
print(df_hi)
df_hi.sum(level='color', axis=1) # in color index level 'Green' are grouped together fi
rst and then sum is applied.
```

```
number      one     three
color     Green Red Green
val1 val2
a    1        0   1     2
     2        3   4     5
b    1        6   7     8
     2        9  10    11
```

Out[85]:

	color	Green	Red
val1	val2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

In [86]:

```
# similarly You can work with other summary statistics methods
# The above technique uses the 'groupby' method working principle, and it is neverthles
s to know in future in this course
```

How To Index With DF's columns?

In [88]:

```
# The idea here is we can move row index into the columns and vice versa
```

In [89]:

```
df_c = pd.DataFrame({'a': range(7), 'b': range(14, 7, -1),
                     'c': ['one', 'one', 'one', 'two', 'two','two', 'two'],
                     'd': [0, 1, 2, 0, 1, 2, 3]})
df_c
```

Out[89]:

	a	b	c	d
0	0	14	one	0
1	1	13	one	1
2	2	12	one	2
3	3	11	two	0
4	4	10	two	1
5	5	9	two	2
6	6	8	two	3

In [90]:

```
# DataFrame’s set_index function will create a new DataFrame using one or more of its c
olumns as the index
# Docstring: Set the DataFrame index (row labels) using one or more existing columns. B
y default yields a new object
pd.DataFrame.set_index?
```

In [94]:

```
# Let's convert the column index into row index
df_si = df_c.set_index(['c', 'd'])
df_si
```

Out[94]:

		a	b
c	**d**		
one	**0**	0	14
	1	1	13
	2	2	12
two	**0**	3	11
	1	4	10
	2	5	9
	3	6	8

In [92]:

```
# if you notice the new DF object, the columns have been removed after new index has be
en formed
# we can keep the columns even after reindexing using 'drop=False' argument by default
 it is set to True
df_c.set_index(['c', 'd'], drop=False)
```

Out[92]:

		a	b	c	d
c	**d**				
one	**0**	0	14	one	0
	1	1	13	one	1
	2	2	12	one	2
two	**0**	3	11	two	0
	1	4	10	two	1
	2	5	9	two	2
	3	6	8	two	3

In [95]:

```
# The 'reset_index' method on the otherhand will do the opposit of 'set_index' method
# Here the hierarchical index levels are moved into the columns
df_si.reset_index()
```

Out[95]:

	c	d	a	b
0	one	0	0	14
1	one	1	1	13
2	one	2	2	12
3	two	0	3	11
4	two	1	4	10
5	two	2	5	9
6	two	3	6	8

How To Combine and Merge Datasets?

In [96]:

```
# pandas.merge connects rows in DataFrames based on one or more keys.
# pandas.concat concatenates or “stacks” together objects along an axis.

# The combine_first instance method enables splicing together overlapping data to fill
 in missing values in one object
# with values from another
```

In [98]:

```
# Docstring: Merge DataFrame objects by performing a database-style join operation by c
olumns or indexes.
pd.DataFrame.merge?
```

In [99]:

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                    'data1': range(6)})
df1
```

Out[99]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

In [100]:

```
df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
                   'data2': range(3)})
df2
```

Out[100]:

	key	data2
0	a	0
1	b	1
2	d	2

Let's see Database-Style DataFrame Joins

In [101]:

```
# 'many to one join'
pd.merge(df1, df2)
# In df1 'b' appers first and then 'a'. Hence the merged object consists of df1 data va
lues that are matched with the df2
# data values. only common rows are considered to merge the data values
```

Out[101]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	5	1
3	a	2	0
4	a	4	0

In [102]:

```
# we can also use Pandas style to merge the DF's
df1.merge(df2)
```

Out[102]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	5	1
3	a	2	0
4	a	4	0

In [103]:

```
# By deafault 'merge' method uses overlaping column names as 'merging keys': here key c
olumn is used as the merging key
# we can also specify it explicitly and it is a good practice always
pd.merge(df1, df2, on='key')
```

Out[103]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	5	1
3	a	2	0
4	a	4	0

In [107]:

```
# Suppose if the DF's column names are different we can use 'left_on' and 'right_on' ar
guments separately
df_l = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'b'],
                     'data1': range(6)})
df_l
```

Out[107]:

	lkey	data1
0	*b*	*0*
1	*b*	*1*
2	*a*	*2*
3	*c*	*3*
4	*a*	*4*
5	*b*	*5*

In [105]:

```
df_r = pd.DataFrame({'rkey': ['a', 'b', 'd'],
                    'data2': range(3)})
df_r
```

Out[105]:

	rkey	data2
0	*a*	*0*
1	*b*	*1*
2	*d*	*2*

In [114]:

```
# Let's merge these DF's by switchng on the left and right keys separately
print('df_l', df_l, end='\n'), print('df_r', df_r, end='\n')
pd.merge(df_l, df_r, left_on='lkey', right_on='rkey')
```

```
df_l   lkey  data1
0    b      0
1    b      1
2    a      2
3    c      3
4    a      4
5    b      5
df_r   rkey  data2
0    a      0
1    b      1
2    d      2
```

Out[114]:

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	5	b	1
3	a	2	a	0
4	a	4	a	0

In [115]:

```
# By default merge does an 'inner' join; the keys in the result are the intersection, o
r the common set found in both tables
# Hence in the above 'merge' operation 'c' and 'd' data and associated values are missi
ng
# This is referred as the 'inner' join: it takes the intersection of the keys for mergi
ng operation
```

In [117]:

```
# The other joins are 'left', 'right' and 'outer' joins. use 'how=right' for 'right' jo
in for example. By default this argument
# is set to 'inner'
pd.merge(df_l, df_r,  left_on='lkey', right_on='rkey', how='outer')

# Now we got the fully merged object with uncommon values and associated data are fille
d with missing value sentinels NaN
```

Out[117]:

	lkey	data1	rkey	data2
0	b	0.0	b	1.0
1	b	1.0	b	1.0
2	b	5.0	b	1.0
3	a	2.0	a	0.0
4	a	4.0	a	0.0
5	c	3.0	NaN	NaN
6	NaN	NaN	d	2.0

In [118]:

```
# Similarly you can do 'left' and 'right' joins by passing them separately
```

In [119]:

```
# Let's see  'many to many merge' operation
```

In [120]:

```
df_m1 = pd.DataFrame({'1key': ['b', 'b', 'a', 'c', 'a', 'b'],
              'data1': range(6)})
df_m1           # 'd' is not present
```

Out[120]:

	1key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

In [122]:

```
df_m2 = pd.DataFrame({'2key': ['a', 'b', 'a', 'b', 'd'],
                      'data2': range(5)})
df_m2                  # 'c' is not present
```

Out[122]:

	2key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

In [124]:

```
pd.merge(df_m1, df_m2, left_on='1key', right_on='2key') # Let's ommit "how='outer'" or
 "how='inner'"
```

Out[124]:

	1key	data1	2key	data2
0	b	0	b	1
1	b	0	b	3
2	b	1	b	1
3	b	1	b	3
4	b	5	b	1
5	b	5	b	3
6	a	2	a	0
7	a	2	a	2
8	a	4	a	0
9	a	4	a	2

In [125]:

```
pd.merge(df_m1, df_m2, left_on='1key', right_on='2key', how='outer') # Let's include "how='outer'" or "how='inner'"
```

Out[125]:

	1key	data1	2key	data2
0	b	0.0	b	1.0
1	b	0.0	b	3.0
2	b	1.0	b	1.0
3	b	1.0	b	3.0
4	b	5.0	b	1.0
5	b	5.0	b	3.0
6	a	2.0	a	0.0
7	a	2.0	a	2.0
8	a	4.0	a	0.0
9	a	4.0	a	2.0
10	c	3.0	NaN	NaN
11	NaN	NaN	d	4.0

In [126]:

```
pd.merge(df_m1, df_m2, left_on='1key', right_on='2key', how='left') # Let's include "how='left'
# Notice here the data 'd' is excluded in the merge operation which is in df_m2
```

Out[126]:

	1key	data1	2key	data2
0	b	0	b	1.0
1	b	0	b	3.0
2	b	1	b	1.0
3	b	1	b	3.0
4	a	2	a	0.0
5	a	2	a	2.0
6	c	3	NaN	NaN
7	a	4	a	0.0
8	a	4	a	2.0
9	b	5	b	1.0
10	b	5	b	3.0

In [127]:

```
pd.merge(df_m1, df_m2, left_on='1key', right_on='2key', how='right') # Let's include "how='right'
# Notice here the data 'c' is excluded in the merge operation which is in df_m1
```

Out[127]:

	1key	data1	2key	data2
0	b	0.0	b	1
1	b	1.0	b	1
2	b	5.0	b	1
3	b	0.0	b	3
4	b	1.0	b	3
5	b	5.0	b	3
6	a	2.0	a	0
7	a	4.0	a	0
8	a	2.0	a	2
9	a	4.0	a	2
10	NaN	NaN	d	4

Let's see How we can merge with Multiple column 'keys' as names

In [132]:

```
dfleft = pd.DataFrame({'key1': ['raga', 'raga', 'anuraga'],
                      'key2': ['one', 'two', 'one'],
                      'lval': [1, 2, 3]})

dfright = pd.DataFrame({'key1': ['raga', 'raga', 'anuraga', 'anuraga'],
                       'key2': ['one', 'one', 'one', 'two'],
                       'rval': [4, 5, 6, 7]})
print('dfleft')
print(dfleft)
print()
print('dfright')
print(dfright)
```

```
dfleft
      key1 key2  lval
0     raga  one     1
1     raga  two     2
2  anuraga  one     3

dfright
      key1 key2  rval
0     raga  one     4
1     raga  one     5
2  anuraga  one     6
3  anuraga  two     7
```

In [133]:

```
# Let's merge these two df's with two key columns 'key1' and 'key2'
pd.merge(dfleft, dfright, on=['key1', 'key2'], how='outer') # 'outer' includes all the
 values and associated data
```

Out[133]:

	key1	key2	lval	rval
0	raga	one	1.0	4.0
1	raga	one	1.0	5.0
2	raga	two	2.0	NaN
3	anuraga	one	3.0	6.0
4	anuraga	two	NaN	7.0

In [134]:

```
pd.merge(dfleft, dfright, on=['key1', 'key2'], how='inner') # 'inner' includes only the
common values and associated data
```

Out[134]:

	key1	key2	lval	rval
0	raga	one	1	4
1	raga	one	1	5
2	anuraga	one	3	6

In [135]:

```
pd.merge(dfleft, dfright, on=['key1', 'key2'], how='left') # 'left' includes priority f
or left values and associated data
```

Out[135]:

	key1	key2	lval	rval
0	raga	one	1	4.0
1	raga	one	1	5.0
2	raga	two	2	NaN
3	anuraga	one	3	6.0

In [136]:

```
pd.merge(dfleft, dfright, on=['key1', 'key2'], how='right') # 'right' includes priority
for right values and associated data
```

Out[136]:

	key1	key2	lval	rval
0	raga	one	1.0	4
1	raga	one	1.0	5
2	anuraga	one	3.0	6
3	anuraga	two	NaN	7

In [138]:

```
# while merging we may face some probelm with the overlapping column names. For ex: le
t's switch on only one column name
print('dfleft')
print(dfleft)
print()
print('dfright')
print(dfright)
pd.merge(dfleft, dfright, on='key1')
```

```
dfleft
      key1 key2  lval
0     raga  one     1
1     raga  two     2
2  anuraga  one     3

dfright
      key1 key2  rval
0     raga  one     4
1     raga  one     5
2  anuraga  one     6
3  anuraga  two     7
```

Out[138]:

	key1	key2_x	lval	key2_y	rval
0	raga	one	1	one	4
1	raga	one	1	one	5
2	raga	two	2	one	4
3	raga	two	2	one	5
4	anuraga	one	3	one	6
5	anuraga	one	3	two	7

In [140]:

```
# We can overcome this by suffixing different key suffixes using 'suffixes' argument in
side the merge operation
pd.merge(dfleft, dfright, on='key1', suffixes=('_left', '_right'))
```

Out[140]:

	key1	key2_left	lval	key2_right	rval
0	raga	one	1	one	4
1	raga	one	1	one	5
2	raga	two	2	one	4
3	raga	two	2	one	5
4	anuraga	one	3	one	6
5	anuraga	one	3	two	7

Merging on Row Index

In [141]:

```
# when merge 'keys' found in DF's row index, we can pass can pass left_index=True or ri
ght_index=True (or both) to indicate
# that the index should be used as the merge key
```

In [146]:

```
dfril = pd.DataFrame({'key1': ['raga', 'raga', 'anuraga', 'raga', 'adiraga'],
                       'lval': range(5)})

dfrir = pd.DataFrame({'rval': [1, 2]},
                          index = ['raga', 'anuraga'])
print('dfril')
print(dfril)
print()
print('dfrir')
print(dfrir)
# notice here that, 'raga' and 'anuraga' are the row index values for dfrir
```

```
dfril
      key1  lval
0     raga     0
1     raga     1
2  anuraga     2
3     raga     3
4  adiraga     4

dfrir
         rval
raga        1
anuraga     2
```

In [147]:

```
# Let's switch on the dfril's key index column and pass the right_index=True to enable
 the right df i.e, dfrir's row index
pd.merge(dfril, dfrir, left_on='key1', right_index=True)
# Notice here the dfrir's row index is taken as priority index and the merging has been
taken place based on those values by
# matching with the dfril's key1 values because that is ON
```

Out[147]:

	key1	lval	rval
0	raga	0	1
1	raga	1	1
3	raga	3	1
2	anuraga	2	2

In [149]:

```
# Let's see what happens if swich on left_index=True
#pd.merge(dfril, dfrir, left_on='key1', left_index=True) # if I run this I'll get 'Type
Error' because left DF has implicit
# integer index by default we can't switch it on.
```

In [150]:

```
# Let's make the union of two DF's index's values by passing how='outer' instead of int
ersecting i.e, selecting common index values
pd.merge(dfril, dfrir, left_on='key1', right_index=True, how='outer')
```

Out[150]:

	key1	lval	rval
0	raga	0	1.0
1	raga	1	1.0
3	raga	3	1.0
2	anuraga	2	2.0
4	adiraga	4	NaN

Let's work with hierarchically indexed DataFrames

In [151]:

```
# In hierachically indexed data the joining is implicitely a multiple key merge
```

In [158]:

```
df_l = pd.DataFrame({'key1': ['raga', 'raga', 'raga', 'anuraga', 'anuraga'],
                     'key2': [2000, 2001, 2002, 2001, 2002],
                     'data': np.arange(5.)})

df_r = pd.DataFrame(np.arange(12).reshape((6, 2)),
                      index=[['anuraga', 'anuraga', 'raga', 'raga', 'raga', 'raga'],
                             [2001, 2000, 2000, 2000, 2001, 2002]],
                      columns=['prog1', 'prog2'])
print('df_l:\n', df_l)
print()
print('df_r:\n', df_r) # use '\n' at the end of string to print the object or data vari
able in the next line
```

```
df_l:
       key1  key2  data
0     raga  2000   0.0
1     raga  2001   1.0
2     raga  2002   2.0
3  anuraga  2001   3.0
4  anuraga  2002   4.0

df_r:
               prog1  prog2
anuraga 2001      0      1
        2000      2      3
raga    2000      4      5
        2000      6      7
        2001      8      9
        2002     10     11
```

In [161]:

```
# Let's handle these DF's with multiple keys
# we need to pass a list of multiple column names as keys to merge on hierarchical inde
x values
# The first priority we are giving is for the keys of df_l using 'left_on' and then we
 are considering df_r's index as
# the merge key for merge operation # notice that the df_r also has two columns in its
 index so as to form an hierarchical index
print('df_l:\n', df_l)
print()
print('df_r:\n', df_r)
pd.merge(df_l, df_r, left_on=['key1', 'key2'], right_index=True)
```

```
df_l:
       key1  key2  data
0     raga  2000   0.0
1     raga  2001   1.0
2     raga  2002   2.0
3  anuraga  2001   3.0
4  anuraga  2002   4.0

df_r:
               prog1  prog2
anuraga 2001      0      1
        2000      2      3
raga    2000      4      5
        2000      6      7
        2001      8      9
        2002     10     11
```

Out[161]:

	key1	key2	data	prog1	prog2
0	raga	2000	0.0	4	5
0	raga	2000	0.0	6	7
1	raga	2001	1.0	8	9
2	raga	2002	2.0	10	11
3	anuraga	2001	3.0	0	1

In [160]:

```
# Let's now consider the duplicate index values with 'how=outer' that is forming an uni
on of index values
print('df_l:\n', df_l)
print()
print('df_r:\n', df_r)
pd.merge(df_l, df_r, left_on=['key1', 'key2'], right_index=True, how='outer')
```

Out[160]:

	key1	key2	data	prog1	prog2
0	raga	2000	0.0	4.0	5.0
0	raga	2000	0.0	6.0	7.0
1	raga	2001	1.0	8.0	9.0
2	raga	2002	2.0	10.0	11.0
3	anuraga	2001	3.0	0.0	1.0
4	anuraga	2002	4.0	NaN	NaN
4	anuraga	2000	NaN	2.0	3.0

In [162]:

```
print('df_l:\n', df_l)
print()
print('df_r:\n', df_r)
pd.merge(df_l, df_r, left_on=['key1', 'key2'], right_index=True, how='left')
```

```
df_l:
      key1  key2  data
0     raga  2000   0.0
1     raga  2001   1.0
2     raga  2002   2.0
3  anuraga  2001   3.0
4  anuraga  2002   4.0

df_r:
              prog1  prog2
anuraga 2001      0      1
        2000      2      3
raga    2000      4      5
        2000      6      7
        2001      8      9
        2002     10     11
```

Out[162]:

	key1	key2	data	prog1	prog2
0	raga	2000	0.0	4.0	5.0
0	raga	2000	0.0	6.0	7.0
1	raga	2001	1.0	8.0	9.0
2	raga	2002	2.0	10.0	11.0
3	anuraga	2001	3.0	0.0	1.0
4	anuraga	2002	4.0	NaN	NaN

In [163]:

```
print('df_l:\n', df_l)
print()
print('df_r:\n', df_r)
pd.merge(df_l, df_r, left_on=['key1', 'key2'], right_index=True, how='right')
```

```
df_l:
       key1  key2  data
0     raga  2000   0.0
1     raga  2001   1.0
2     raga  2002   2.0
3  anuraga  2001   3.0
4  anuraga  2002   4.0

df_r:
               prog1  prog2
anuraga 2001      0      1
        2000      2      3
raga    2000      4      5
        2000      6      7
        2001      8      9
        2002     10     11
```

Out[163]:

	key1	key2	data	prog1	prog2
0	raga	2000	0.0	4	5
0	raga	2000	0.0	6	7
1	raga	2001	1.0	8	9
2	raga	2002	2.0	10	11
3	anuraga	2001	3.0	0	1
4	anuraga	2000	NaN	2	3

In [164]:

```
# suppose if we have a multiple index on both the DF's then merging is possible just by switching on the index values of both
# the DF's using 'left_index=True' and 'right_index=True'
```

In [165]:

```
df_li = pd.DataFrame([[10, 20], [30, 40], [50, 60]],
                      index=['a', 'c', 'e'],
                      columns=['raga', 'anuraga'])

df_ri = pd.DataFrame([[70, 80], [90, 100], [110, 120], [130, 140]],
                       index=['b', 'c', 'd', 'e'],
                       columns=['braga', 'sraga'])
print('df_li\n', df_li)
print()
print('df_ri\n', df_ri)
```

```
df_li
    raga  anuraga
a     10       20
c     30       40
e     50       60

df_ri
    braga  sraga
b     70     80
c     90    100
d    110    120
e    130    140
```

In [166]:

```
pd.merge(df_li, df_ri, how='outer', left_index=True, right_index=True)
```

Out[166]:

	raga	anuraga	braga	sraga
a	10.0	20.0	NaN	NaN
b	NaN	NaN	70.0	80.0
c	30.0	40.0	90.0	100.0
d	NaN	NaN	110.0	120.0
e	50.0	60.0	130.0	140.0

In [167]:

```
pd.merge(df_li, df_ri, how='inner', left_index=True, right_index=True)
```

Out[167]:

	raga	anuraga	braga	sraga
c	30	40	90	100
e	50	60	130	140

In [168]:

```
# similarly you can check for the remaining cases: left and right and then analyze what
happens
```

In [169]:

```
# DF has a very good option to join the DF's of same or similar indexes with non-overla
pping column names:
# The method is 'join'
pd.DataFrame.join?
```

In [170]:

```
df_li.join(df_ri, how='outer')
```

Out[170]:

	raga	anuraga	braga	sraga
a	10.0	20.0	NaN	NaN
b	NaN	NaN	70.0	80.0
c	30.0	40.0	90.0	100.0
d	NaN	NaN	110.0	120.0
e	50.0	60.0	130.0	140.0

In [172]:

```
# suppose if the two DF's have similar column names with different data values, the joi
n method has 'lsuffix' and 'rsuffix'
# to differentiate them
# Let's see the simple example
caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
                       'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
                      'B': ['B0', 'B1', 'B2']})
print('caller:\n', caller)
print()
print('other:\n', other)
```

```
caller:
   key   A
0  K0  A0
1  K1  A1
2  K2  A2
3  K3  A3
4  K4  A4
5  K5  A5

other:
   key   B
0  K0  B0
1  K1  B1
2  K2  B2
```

In [173]:

```
# Since both the DF's have the similar columns with the common column name 'key'
# Let's use suffixes to use from both left DF and right DF
caller.join(other, lsuffix='_caller', rsuffix='_other') # default how in join is left,
 we can make it other
```

Out[173]:

	key_caller	A	key_other	B
0	K0	A0	K0	B0
1	K1	A1	K1	B1
2	K2	A2	K2	B2
3	K3	A3	NaN	NaN
4	K4	A4	NaN	NaN
5	K5	A5	NaN	NaN

In [174]:

```
caller.join(other, lsuffix='_caller', rsuffix='_other', how='right')
```

Out[174]:

	key_caller	A	key_other	B
0	K0	A0	K0	B0
1	K1	A1	K1	B1
2	K2	A2	K2	B2

In [175]:

```
# Join method also accepts the DF's which are modified to have the same index name with
'set_index' method
# Let's set 'key' column of both the DF's as their index
caller.set_index('key').join(other.set_index('key'))
```

Out[175]:

	A	B
key		
K0	A0	B0
K1	A1	B1
K2	A2	B2
K3	A3	NaN
K4	A4	NaN
K5	A5	NaN

In [176]:

```
# we can also make one of the DF's column as the index and other DF's column as the mer
ging key using 'on' argument
# Let's make 'other' DF's column 'key' as the index and switch on the merge key(column
 name) of 'caller' DF
caller.join(other.set_index('key'), on='key')
```

Out[176]:

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	NaN
4	K4	A4	NaN
5	K5	A5	NaN

In [177]:

```
# Join method also accepts a list of DF's to join them, similar to a 'concat' operation
which concats two or more DF's.
```

How To Concatenate DataFrame's Along The Row or Column Axis?

In [178]:

```
#
```

In [181]:

```
#
pd.concat?
```

Docstring: Concatenate pandas objects along a particular axis with optional set logic along the other axes. Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.

In [182]:

```
ser1 = pd.Series([0, 1], index=['A', 'B'])

ser2 = pd.Series([2, 3, 4], index=['C', 'D', 'E'])

ser3 = pd.Series([5, 6], index=['F', 'G'])
print(ser1); print(ser2); print(ser3)
```

```
A    0
B    1
dtype: int64
C    2
D    3
E    4
dtype: int64
F    5
G    6
dtype: int64
```

In [184]:

```
# By default 'concat' function works along axis=0, that is along row index
# so concatenation of series produces another series along an axis=0
pd.concat([ser1, ser2, ser3])
```

Out[184]:

```
A    0
B    1
C    2
D    3
E    4
F    5
G    6
dtype: int64
```

In [221]:

```
# Let's change the axis to concat along the column index
pd.concat([ser1, ser2, ser3], axis=1, sort=True) # with sort=False, that is index will
 be sorted or un sorted or kept as it is
```

Out[221]:

	0	1	2
A	0.0	NaN	NaN
B	1.0	NaN	NaN
C	NaN	2.0	NaN
D	NaN	3.0	NaN
E	NaN	4.0	NaN
F	NaN	NaN	5.0
G	NaN	NaN	6.0

In [222]:

```
# concatenating along an axis='column' produces a DF with non-overlapping values filled
by NaN
# which is similar to like joining with the union of index, that is how='outer' in case
of merge or join
print(ser1); print(ser2); print(ser3)
pd.concat([ser1, ser2, ser3], axis=1, sort=True) # with sort=False
```

```
A    0
B    1
dtype: int64
C    2
D    3
E    4
dtype: int64
F    5
G    6
dtype: int64
```

Out[222]:

	0	1	2
A	0.0	NaN	NaN
B	1.0	NaN	NaN
C	NaN	2.0	NaN
D	NaN	3.0	NaN
E	NaN	4.0	NaN
F	NaN	NaN	5.0
G	NaN	NaN	6.0

In [223]:

```
# we can concatenate by intersecting the indexes using 'join=inner' argument
# in this example it results null concatenation because of non-overlapping indexes
pd.concat([ser1, ser2, ser3], axis=1, sort=True, join='inner')
```

Out[223]:

0	1	2

In [224]:

```
# Let's create a Series with overlapping indexes and join
ser4 = pd.concat([ser1, ser3])
ser4
```

Out[224]:

```
A    0
B    1
F    5
G    6
dtype: int64
```

In [225]:

```
print(ser1); print(ser4)
pd.concat([ser1, ser4], axis=1, sort=True) # this is the outer join, by default join=ou
ter in concat function
```

```
A    0
B    1
dtype: int64
A    0
B    1
F    5
G    6
dtype: int64
```

Out[225]:

	0	1
A	0.0	0
B	1.0	1
F	NaN	5
G	NaN	6

In [226]:

```
print(ser1); print(ser4)
pd.concat([ser1, ser4], axis=1, join='inner') # only intersected columns are combined t
ogether
```

```
A    0
B    1
dtype: int64
A    0
B    1
F    5
G    6
dtype: int64
```

Out[226]:

	0	1
A	0	0
B	1	1

In [227]:

```
# we can specify axes to be used on the other axes with 'join_axex' argument, this is s
imilar to like outer join
pd.concat([ser1, ser4], axis=1, join_axes=[['A', 'B', 'F', 'G']])
```

Out[227]:

	0	1
A	0.0	0
B	1.0	1
F	NaN	5
G	NaN	6

In [228]:

```
# The concatented pices are not idenfiable in the result by default. however we can do
 so using keys argument
# this will create an hierarchical index to identify the objects used to concatenate
idc = pd.concat([ser1, ser2, ser3], axis=0, keys=['1', '2', '3'])
idc
# Here the '1', '2' and '3' identifies the three series that concatenated just now. You
can also strings to label the keys
```

Out[228]:

```
1  A    0
   B    1
2  C    2
   D    3
   E    4
3  F    5
   G    6
dtype: int64
```

In [229]:

```
# unstack() method on this concatenation makes keys as the new index label, it is simil
ar to like unstacking the
# hierarchical index
idc.unstack()
```

Out[229]:

	A	B	C	D	E	F	G
1	0.0	1.0	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	2.0	3.0	4.0	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	5.0	6.0

In [230]:

```
# we can use keys to concatenate along axis=1, here the keys becomes the DF's columns
pd.concat([ser1, ser2, ser3], axis=1, keys=['1', '2', '3'], sort=True)
```

Out[230]:

	1	2	3
A	0.0	NaN	NaN
B	1.0	NaN	NaN
C	NaN	2.0	NaN
D	NaN	3.0	NaN
E	NaN	4.0	NaN
F	NaN	NaN	5.0
G	NaN	NaN	6.0

Let's see the same logic on DataFrame objects

In [251]:

```
df_li = pd.DataFrame([[10, 20], [30, 40], [50, 60]],
                      index=['a', 'c', 'e'],
                      columns=['raga', 'anuraga'])

df_ri = pd.DataFrame([[70, 80], [90, 100], [110, 120], [130, 140]],
                       index=['b', 'c', 'd', 'e'],
                       columns=['braga', 'sraga'])
print('df_li\n', df_li)
print()
print('df_ri\n', df_ri)
```

```
df_li
    raga  anuraga
a    10       20
c    30       40
e    50       60

df_ri
    braga  sraga
b     70     80
c     90    100
d    110    120
e    130    140
```

In [254]:

```
# Join=inner combines only the common columns along column index
pd.concat([df_li, df_ri], axis=1, keys=['one', 'two'], sort=True, join='inner')
```

Out[254]:

	one		two	
	raga	anuraga	braga	sraga
c	30	40	90	100
e	50	60	130	140

In [253]:

```
# join='outer' takes union of the index values to combine the DF's
pd.concat([df_li, df_ri], axis=1, keys=['1', '2', '3'], sort=True, join='outer') # noti
ce that the index values are sorted
```

Out[253]:

	1		2	
	raga	anuraga	braga	sraga
a	10.0	20.0	NaN	NaN
b	NaN	NaN	70.0	80.0
c	30.0	40.0	90.0	100.0
d	NaN	NaN	110.0	120.0
e	50.0	60.0	130.0	140.0

In [255]:

```
# we can pass dictionary keys for the index levels,
pd.concat({'level1':df_li, 'level2':df_ri}, sort=True, join='outer') # along axis=0
```

Out[255]:

		anuraga	braga	raga	sraga
level1	a	20.0	NaN	10.0	NaN
	c	40.0	NaN	30.0	NaN
	e	60.0	NaN	50.0	NaN
level2	b	NaN	70.0	NaN	80.0
	c	NaN	90.0	NaN	100.0
	d	NaN	110.0	NaN	120.0
	e	NaN	130.0	NaN	140.0

In [256]:

```
pd.concat({'level1':df_li, 'level2':df_ri}, sort=True, join='outer', axis=1) # along ax
is=1
```

Out[256]:

	level1		level2	
	raga	anuraga	braga	sraga
a	10.0	20.0	NaN	NaN
b	NaN	NaN	70.0	80.0
c	30.0	40.0	90.0	100.0
d	NaN	NaN	110.0	120.0
e	50.0	60.0	130.0	140.0

In [257]:

```
# we can also name the created column axis levels using names argument
pd.concat({'level1':df_li, 'level2':df_ri}, sort=True, join='outer', axis=1, names=['fi
rst', 'second'])
```

Out[257]:

first	level1		level2	
second	raga	anuraga	braga	sraga
a	10.0	20.0	NaN	NaN
b	NaN	NaN	70.0	80.0
c	30.0	40.0	90.0	100.0
d	NaN	NaN	110.0	120.0
e	50.0	60.0	130.0	140.0

In [258]:

```
# what happens if the DF's doesn't have any row index levels or if row index does not c
ontain any relevant data
df_li = pd.DataFrame([[10, 20], [30, 40], [50, 60]],
                      columns=['raga', 'anuraga'])

df_ri = pd.DataFrame([[70, 80], [90, 100], [110, 120], [130, 140]],
                       columns=['braga', 'sraga'])
print('df_li\n', df_li)
print()
print('df_ri\n', df_ri)
```

```
df_li
    raga  anuraga
0    10       20
1    30       40
2    50       60

df_ri
    braga  sraga
0     70     80
1     90    100
2    110    120
3    130    140
```

In [262]:

```
# in that case we pass 'ignore_index=True' argument
pd.concat([df_li, df_ri], axis=0, join='outer', ignore_index=True, sort=True)
```

Out[262]:

	anuraga	braga	raga	sraga
0	20.0	NaN	10.0	NaN
1	40.0	NaN	30.0	NaN
2	60.0	NaN	50.0	NaN
3	NaN	70.0	NaN	80.0
4	NaN	90.0	NaN	100.0
5	NaN	110.0	NaN	120.0
6	NaN	130.0	NaN	140.0

In [264]:

```
pd.concat([df_li, df_ri], axis=1, join='outer', ignore_index=True, sort=True)
```

Out[264]:

	0	1	2	3
0	10.0	20.0	70	80
1	30.0	40.0	90	100
2	50.0	60.0	110	120
3	NaN	NaN	130	140

In [265]:

```
# You can easily work on remaining arguments of the concat function
```

How To Combine Data With Overlap?

combine_first

Docstring: Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

In [ ]:

```
pd.DataFrame.combine_first?
```

In [270]:

```
df1 = pd.DataFrame([[1, np.nan]])
df2 = pd.DataFrame([[3, 4]])
print('df1:\n', df1)
print('df2:\n', df2)
```

```
df1:
    0   1
0  1 NaN
df2:
    0  1
0  3  4
```

In [271]:

```
df1.combine_first(df2)
```

Out[271]:

	0	1
0	1	4.0

In [276]:

```
# There is one more function 'combine': Add two DataFrame objects and do not propagate
 NaN values, so if for a (column, time)
# one frame is missing a value, it will default to the other frame's value (which might
be NaN as well)
pd.DataFrame.combine?
```

In [279]:

```
df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
print('df1:\n', df1)
print('df2:\n', df2)
```

```
df1:
    A  B
0  0  4
1  0  4
df2:
    A  B
0  1  3
1  1  3
```

In [280]:

```
df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
```

Out[280]:

	A	B
0	*0*	*3*
1	*0*	*3*

In [281]:

```
# if you observe the combine_first method which works like an if-else function
# where as combine method takes a function with non-null values and add two DF's based
 on the function passed
```

How To Reshape and Pivot Pandas Data?

In [282]:

```
# Pandas provides many ways to rearrange the Tabular Data and is known as 'reshape or p
ivot' operation
```

Reshaping The Hierarchically indexed DataFrame's

In [283]:

```
# 'stack' and 'unstack' methods are the two very important method to do this operation
# Docstring: Stack the prescribed level(s) from columns to index.
# Return a reshaped DataFrame or Series having a multi-level index with one or more new
inner-most levels compared to the
# current DataFrame
pd.DataFrame.stack?
```

In [284]:

```
df_s = data = pd.DataFrame(np.arange(6).reshape((2, 3)),
                     index=pd.Index(['raga', 'mmraga'], name='state'),
                     columns=pd.Index(['one', 'two', 'three'], name='number'))
df_s
```

Out[284]:

number	one	two	three
state			
raga	0	1	2
mmraga	3	4	5

In [286]:

```
# calling stack on this method pivotes columns into rows
df_s.stack()
```

Out[286]:

```
state   number
raga    one       0
        two       1
        three     2
mmraga  one       3
        two       4
        three     5
dtype: int32
```

In [288]:

```
# Docstring: Pivot a level of the (necessarily hierarchical) index labels, returning a
 DataFrame having a new level of
# column labels whose inner-most level consists of the pivoted index labels. If the ind
ex is not a MultiIndex,the output
# will be a Series (the analogue of stack when the columns are not a MultiIndex).The le
vel involved will automatically get sorted.
pd.DataFrame.unstack?
```

In [287]:

```
# calling unstack on stacked object will reverse the operation of stack
df_s.stack().unstack() # the default level=-1
```

Out[287]:

number	one	two	three
state			
raga	0	1	2
mmraga	3	4	5

In [289]:

```
# The default unstack is on the innermost level, we can unstack on different level by p
assing level name or number
df_s.stack().unstack(level=0) # the column index is considered to unstack the data
```

Out[289]:

state	raga	mmraga
number		
one	0	3
two	1	4
three	2	5

In [290]:

```
# we can also name instead
df_s.stack().unstack('state')
```

Out[290]:

state	raga	mmraga
number		
one	0	3
two	1	4
three	2	5

In [300]:

```
# suppose if the subgroups objects ara not having the same values in all the index leve
ls, then missing values will be introduced.
s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'], name='one')
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'], name='two')
data = pd.concat([s1, s2], keys= ['one', 'two'])
data
```

Out[300]:

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

In [301]:

```
data.unstack()
```

Out[301]:

	a	b	c	d	e
one	0.0	1.0	2.0	3.0	NaN
two	NaN	NaN	4.0	5.0	6.0

In [302]:

```
# stack method on this removes the missing data and revert back to the original shape
data.unstack().stack()
```

Out[302]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64
```

In [303]:

```
# we can pass 'dropna=False' to hold the missing values in the stack method if needed
data.unstack().stack(dropna=False)
```

Out[303]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

In [304]:

```
# Important to remember: stack()has 'dropna=True' by default where as unstack() has fil
l_value=None by default
```

In [315]:

```
# When you unstack the DF's index the unstacked level will the lowest level in the retu
rning object
```

In [311]:

```
df_s.stack().unstack('state')
```

Out[311]:

state	raga	mmraga
number		
one	0	3
two	1	4
three	2	5

In [ ]:

```

```