Module 5

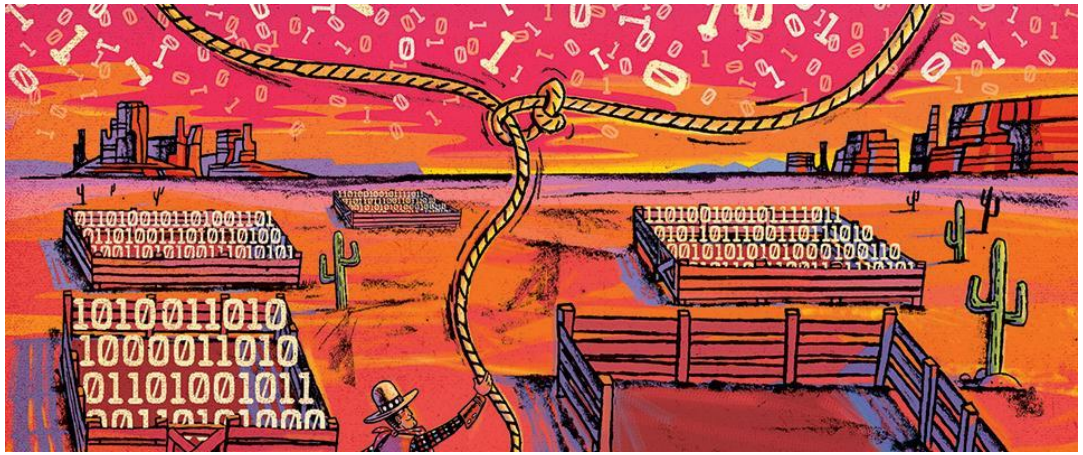
Use Case 1

LESSON 1

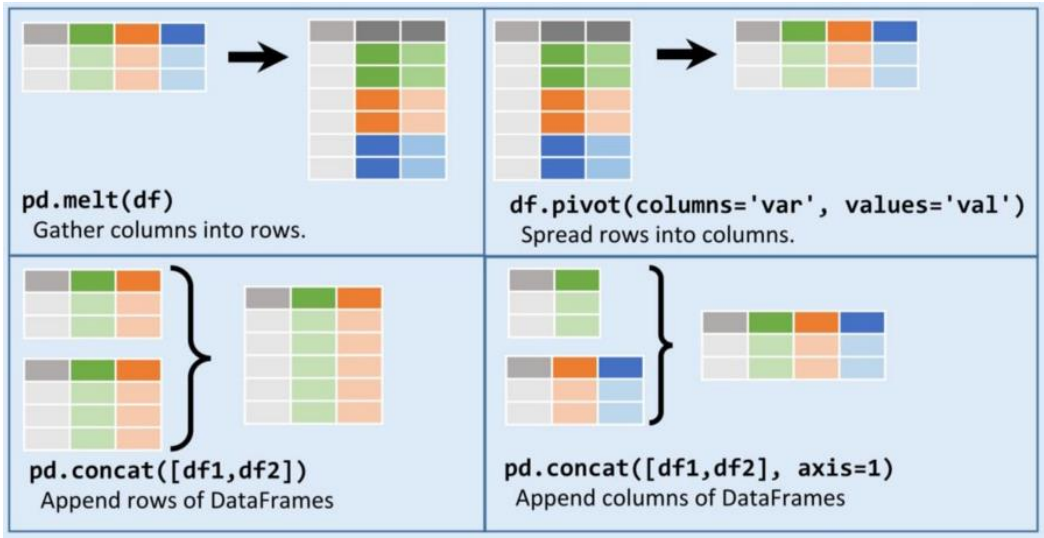
Data Preparation: Cleaning and **Wrangling**

José Manuel García Nieto – University of Málaga

- The context of data analytics
  - Any data project needs a previous step to be successful, which comprises two main tasks:
    - Data Cleaning
    - Data Wrangling
  - In fact, they are usually the most time consuming for data analysts
    - According to a survey conducted in 2017, a data analyst can spend, on average, 80% of their time on Data Wrangling.

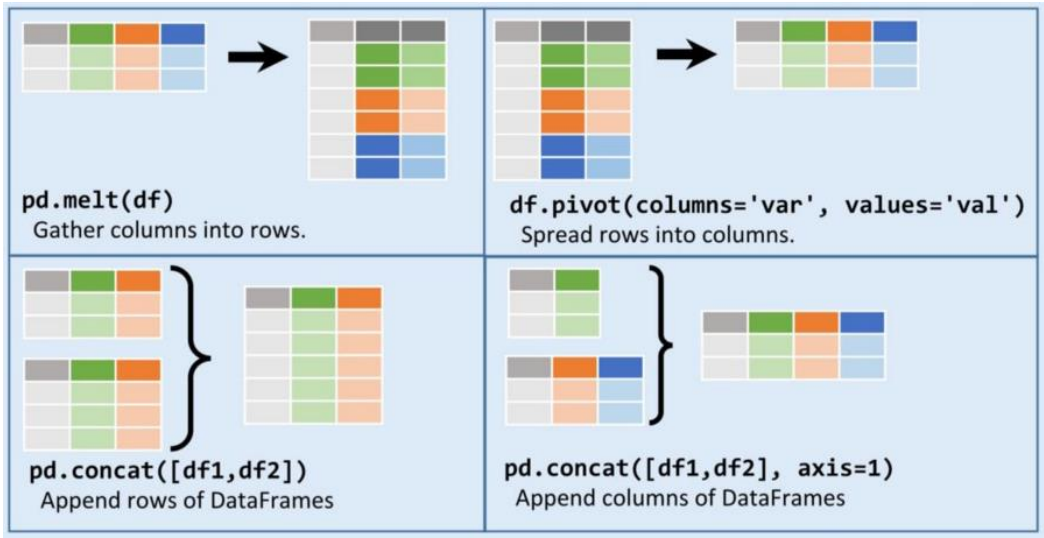


- Data Wrangling: Join, Combine, and Reshape
  - Hierarchical Indexing
    - Reordering and Sorting Levels
    - Summary Statistics by Level
    - Indexing with a DataFrame's columns
  - Combining and Merging Datasets
    - Database-Style DataFrame Joins
    - Merging on Index
    - Concatenating Along an Axis
    - Combining Data with Overlap
  - Reshaping and Pivoting
    - Reshaping with Hierarchical Indexing
    - Pivoting "Long" to "Wide" Format
    - Pivoting "Wide" to "Long" Format



- Data Wrangling: Join, Combine, and Reshape

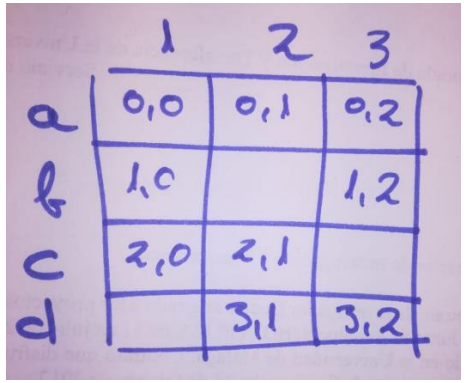
  - Hierarchical Indexing
    - Reordering and Sorting Levels
    - Summary Statistics by Level
    - Indexing with a DataFrame's columns
  - Combining and Merging Datasets
    - Database-Style DataFrame Joins
    - Merging on Index
    - Concatenating Along an Axis
    - Combining Data with Overlap
  - Reshaping and Pivoting
    - Reshaping with Hierarchical Indexing
    - Pivoting "Long" to "Wide" Format
    - Pivoting "Wide" to "Long" Format



- Hierarchical Indexing
  - Hierarchical indexing in pandas is used extensively in some wrangling operations
  - It enables to have multiple (two or more) index levels on an axis
  - It provides a way to work with higher dimensional data in a lower dimensional form

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                 index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...:                        [1, 2, 3, 1, 3, 1, 2, 2, 3]])
```

```
In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```



Series with a **MultiIndex** as its index

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

- Hierarchical Indexing
  - Hierarchically indexed object, so-called partial indexing is possible, enabling to concisely select subsets of the data

```
In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64
```

```
In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64
```

```
In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
d  2    0.281746
   3    0.769023
dtype: float64
```

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

- Hierarchical Indexing
  - It plays an important role in reshaping data and group-based operations like forming a pivot table
  - For example, rearrange the data into a DataFrame using its **unstack** method

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

  - The inverse operation of unstack is **stack**

- Hierarchical Indexing
  - With a DataFrame, either axis can have a hierarchical index

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
    ....:                     index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
    ....:                     columns=[['Ohio', 'Ohio', 'Colorado'],
    ....:                              ['Green', 'Red', 'Green']])
```

```
In [19]: frame
Out[19]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

  - The hierarchical levels can have names (as strings or any Python objects)

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state       Ohio     Colorado
color      Green Red    Green
key1 key2
a    1         0   1        2
     2         3   4        5
b    1         6   7        8
     2         9  10       11
```

With partial column indexing you can similarly select groups of columns

```
In [23]: frame['Ohio']
Out[23]:
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

- Hierarchical Indexing: Reordering and Sorting Levels
  - To rearrange the order of the levels on an axis or sort the data by the values in one specific level
    - The **swaplevel** takes two level numbers or names and returns a new object with the levels interchanged
    - **sort_index** sorts the data using only the values in a single level

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
2    a        3   4        5
1    b        6   7        8
2    b        9  10       11
```

```
In [25]: frame.sort_index(level=1)
Out[25]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
b    1        6   7        8
a    2        3   4        5
b    2        9  10       11
```

```
In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
     b        6   7        8
2    a        3   4        5
     b        9  10       11
```

Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level, calling

sort_index(level=0)
or
sort_index().

- Hierarchical Indexing: Summary Statistics by Level
  - Many descriptive and summary statistics on DataFrame and Series have a level option in which to specify the level to aggregate by on a particular axis

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

```
In [27]: frame.sum(level='key2')
Out[27]:
state  Ohio     Colorado
color Green Red    Green
key2
1          6   8       10
2         12  14       16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

frame.mean(level='key2')

- Hierarchical Indexing: Indexing with a DataFrame's columns
  - Sometimes, it is required to use one or more columns from a DataFrame as the row index

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
    ....:                      'c': ['one', 'one', 'one', 'two', 'two',
    ....:                            'two', 'two'],
    ....:                      'd': [0, 1, 2, 0, 1, 2, 3]})
```

```
In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

  - DataFrame's **set_index** function will create a new DataFrame using one or more of its columns as the index

```
In [31]: frame2 = frame.set_index(['c', 'd'])
```

```
In [32]: frame2
Out[32]:
       a  b
c   d
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

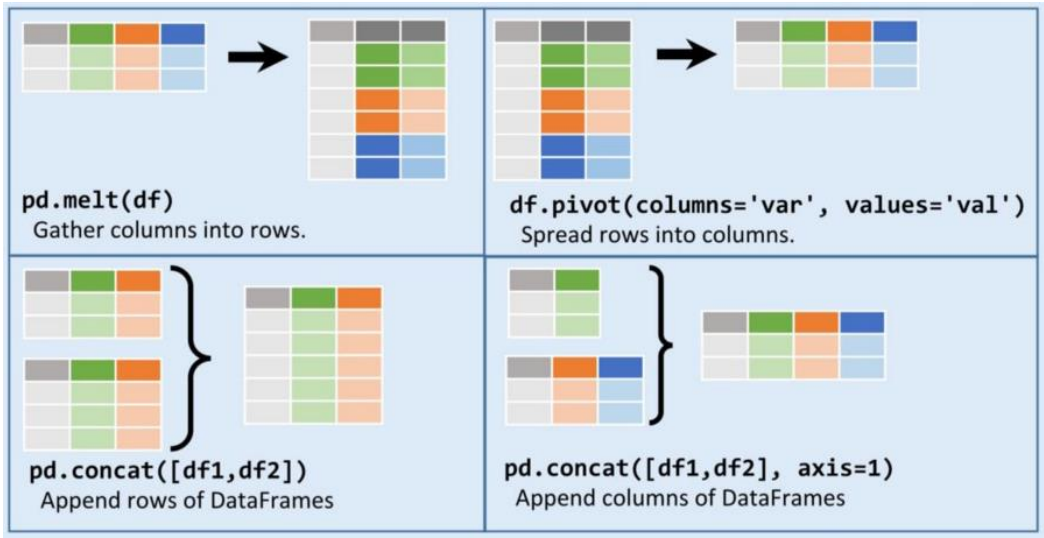
```
In [33]: frame.set_index(['c', 'd'], drop=False)
```

```
Out[33]:
         a  b    c  d
c   d
one 0    0  7  one  0
    1    1  6  one  1
    2    2  5  one  2
two 0    3  4  two  0
    1    4  3  two  1
    2    5  2  two  2
    3    6  1  two  3
```

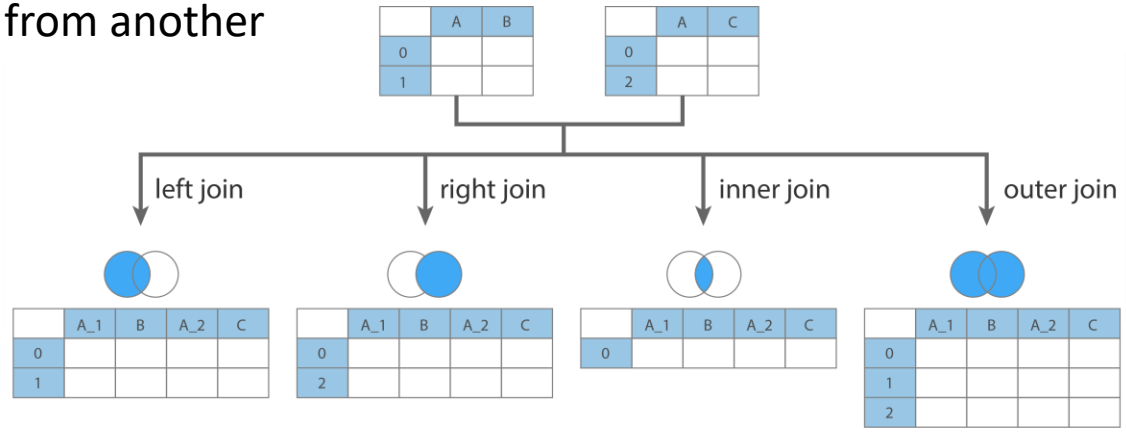
  - **reset_index**, does the opposite of **set_index**

```
In [34]: frame2.reset_index()
```

- Data Wrangling: Join, Combine, and Reshape
  - Hierarchical Indexing
    - Reordering and Sorting Levels
    - Summary Statistics by Level
    - Indexing with a DataFrame's columns
  - Combining and Merging Datasets
    - Database-Style DataFrame Joins
    - Merging on Index
    - Concatenating Along an Axis
    - Combining Data with Overlap
  - Reshaping and Pivoting
    - Reshaping with Hierarchical Indexing
    - Pivoting "Long" to "Wide" Format
    - Pivoting "Wide" to "Long" Format

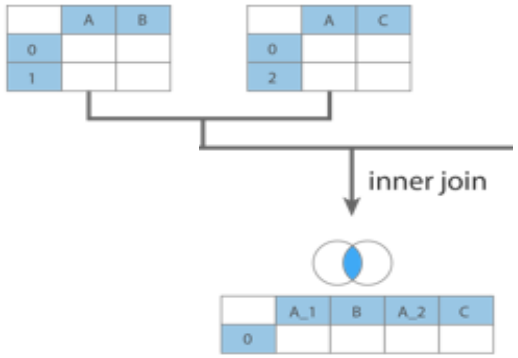


- Combining and Merging Datasets
  - Data contained in pandas objects can be combined together in a number of ways
    - **pandas.merge** connects rows in DataFrames based on one or more keys (= joins in SQL)
    - **pandas.concat** concatenates or “stacks” together objects along an axis
    - **combine_first** instance method enables splicing together overlapping data
      - to fill in missing values in one object with values from another



- Combining and Merging Datasets: Database-Style DataFrame Joins
  - Merge or join operations combine datasets by linking rows using one or more keys (SQL-based operations)

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
    ....:                    'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
    ....:                    'data2': range(3)})
```




```
In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b
```

```
In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

many-to-one join

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

```
In [40]: pd.merge(df1, df2, on='key')
```

Making the overlapping column explicit

- Combining and Merging Datasets: Database-Style DataFrame Joins
  - If the column names are different in each object, then specify them separately

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
    ....:                   'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
    ....:                   'data2': range(3)})
```

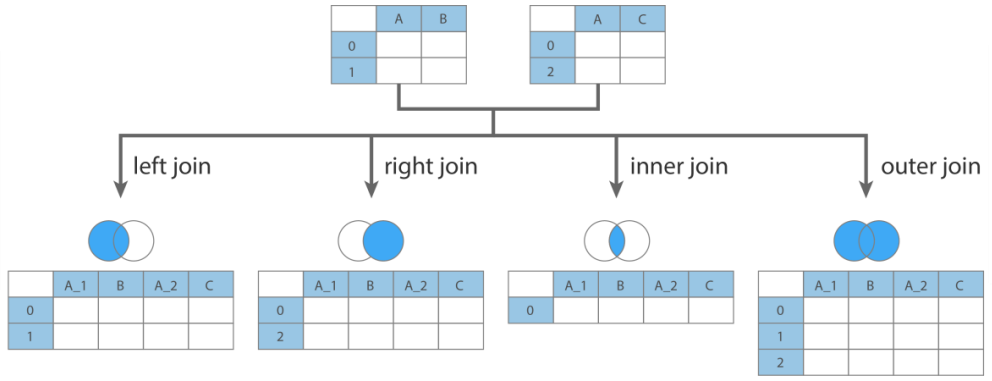
```
In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

'c' and 'd' values and associated data are missing from the result

Default: inner join

  - Other possible options are 'left', 'right', and 'outer'
    - The outer join takes the union of the keys, combining the effect of applying both left and right joins

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key  data2
0    0.0   b    1.0
1    1.0   b    1.0
2    6.0   b    1.0
3    2.0   a    0.0
4    4.0   a    0.0
5    5.0   a    0.0
6    3.0   c    NaN
7    NaN   d    2.0
```



- Combining and Merging Datasets: Database-Style DataFrame Joins
  - To merge with multiple keys, pass a list of column names

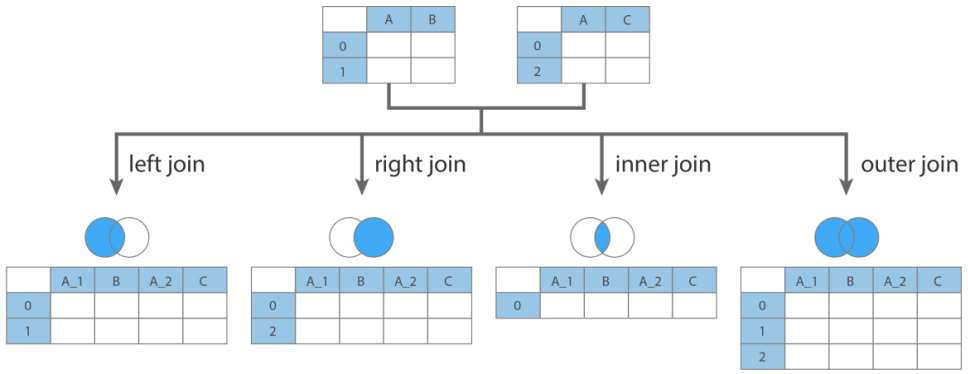
```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                     'key2': ['one', 'two', 'one'],
   ....:                     'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                      'key2': ['one', 'one', 'one', 'two'],
   ....:                      'rval': [4, 5, 6, 7]})
```

```
In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
Out[53]:
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

  - To treat with overlapping column names merge has a suffixes option

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  key1 key2_x  lval key2_y  rval
0  foo    one     1    one     4
1  foo    one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
4  bar    one     3    one     6
5  bar    one     3    two     7
```

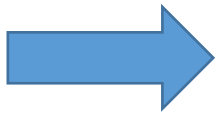


- Combining and Merging Datasets: Merging on Index
  - In some cases, the merge key(s) in a DataFrame will be found in its index
  - In this case, let pass **left_index=True** or **right_index=True** (or both) to indicate that the index should be used as the merge key

```
In [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                       'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

```
In [58]: left1
Out[58]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5
```

```
In [59]: right1
Out[59]:
   group_val
a        3.5
b        7.0
```



```
In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

Intersection,
with the inner join

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[61]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

Or union,
with the outer join

- Combining and Merging Datasets: Merging on Index
  - With hierarchically indexed data, joining on index is implicitly a multiple-key merge

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
   ....:                               'Nevada', 'Nevada'],
   ....:                      'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                      'data': np.arange(5.)})

In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
   ....:                       index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',
   ....:                               'Ohio', 'Ohio'],
   ....:                              [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                       columns=['event1', 'event2'])
```

```
In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002
```

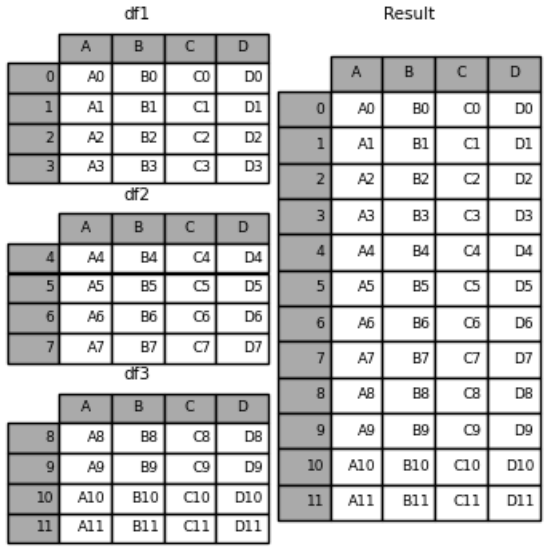
```
In [65]: righth
Out[65]:
             event1  event2
Nevada 2001       0       1
       2000       2       3
Ohio   2000       4       5
       2000       6       7
       2001       8       9
       2002      10      11
```

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[66]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000       4       5
0   0.0    Ohio  2000       6       7
1   1.0    Ohio  2001       8       9
2   2.0    Ohio  2002      10      11
3   3.0  Nevada  2001       0       1
```

```
In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:          right_index=True, how='outer')
Out[67]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000     4.0     5.0
0   0.0    Ohio  2000     6.0     7.0
1   1.0    Ohio  2001     8.0     9.0
2   2.0    Ohio  2002    10.0    11.0
3   3.0  Nevada  2001     0.0     1.0
4   4.0  Nevada  2002     NaN     NaN
4   NaN  Nevada  2000     2.0     3.0
```

Or union, with the outer join

- Combining and Merging Datasets: Concatenating Along an Axis
  - Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking
    - Thinks to take into account:
      - If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?
      - Do the concatenated chunks of data need to be identifiable in the resulting object?
      - Does the “concatenation axis” contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation



- Combining and Merging Datasets: Concatenating Along an Axis
  - The **concat** function in pandas provides a consistent way to address each of these concerns. Imagine 3 series:

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

  - Calling concat with these objects in a list glues together the values and indexes

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
     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
```

On horizontal axis=1

On overlapping indexes, it is possible to intersect by passing ***join='inner'***

- Combining and Merging Datasets: Concatenating Along an Axis
  - A potential issue is that the concatenated pieces are not identifiable in the result (e. g. batching in streaming processing from sensors)
  - To solve this, create a hierarchical index on the concatenation axis by using the keys argument

```
In [92]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [93]: result
Out[93]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64
```

```
In [94]: result.unstack()
Out[94]:
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

- Combining and Merging Datasets: Concatenating Along an Axis
  - A last consideration concerns DataFrames in which the row index does not contain any relevant data

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
```

```
In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741

In [106]: df2
Out[106]:
          b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

  - In this case, you can pass **ignore_index=True**

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985       NaN  3.248944
4  0.302614 -0.577087       NaN  0.124121
```

- Combining and Merging Datasets: Combining Data with Overlap
  - With DataFrames, **combine_first** function “patchs” missing data in the calling object with data from the passed one

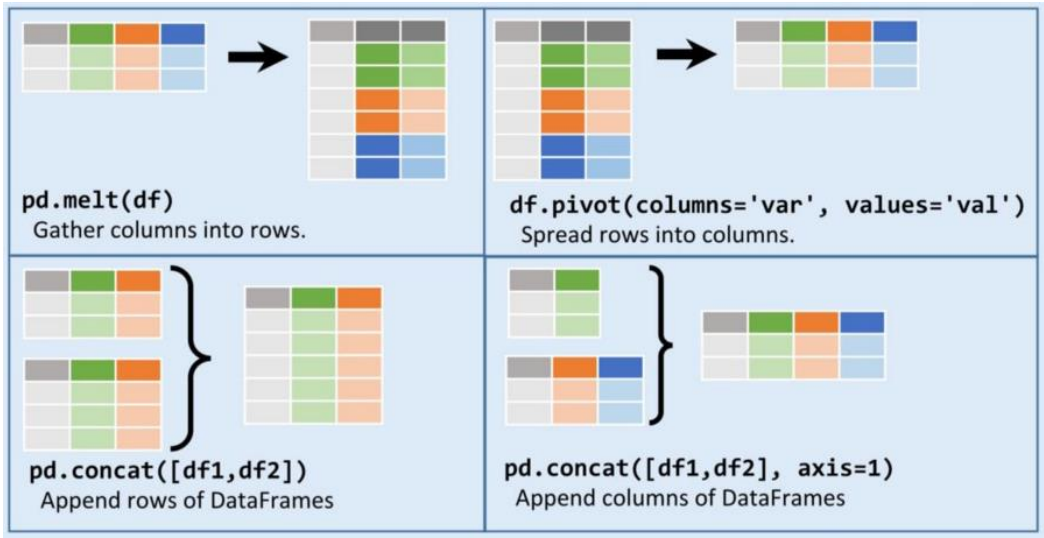
```
In [115]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
   .....:                     'b': [np.nan, 2., np.nan, 6.],
   .....:                     'c': range(2, 18, 4)})

In [116]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
   .....:                     'b': [np.nan, 3., 4., 6., 8.]})
```

```
In [117]: df1
Out[117]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14
```

```
In [118]: df2
Out[118]:
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0
```

```
In [119]: df1.combine_first(df2)
Out[119]:
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

- Data Wrangling: Join, Combine, and Reshape
  - Hierarchical Indexing
    - Reordering and Sorting Levels
    - Summary Statistics by Level
    - Indexing with a DataFrame's columns
  - Combining and Merging Datasets
    - Database-Style DataFrame Joins
    - Merging on Index
    - Concatenating Along an Axis
    - Combining Data with Overlap
  - Reshaping and Pivoting
    - Reshaping with Hierarchical Indexing
    - Pivoting "Long" to "Wide" Format
    - Pivoting "Wide" to "Long" Format



- Reshaping and Pivoting: Reshaping with Hierarchical Indexing
  - Hierarchical indexing provides a consistent way to rearrange data in a DataFrame
  - There are two primary actions:
    - **stack:** This "rotates" or pivots from the columns in the data to the rows
    - **unstack**: This pivots from the rows into the columns

```
In [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....:                     index=pd.Index(['Ohio', 'Colorado'], name='state'),
   .....:                     columns=pd.Index(['one', 'two', 'three'],
   .....:                     name='number'))
```

```
In [121]: data
Out[121]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

```
In [122]: result = data.stack()

In [123]: result
Out[123]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

```
In [124]: result.unstack()
Out[124]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

By level: 'state'

```
In [126]: result.unstack('state')
Out[126]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

- Reshaping and Pivoting: Reshaping with Hierarchical Indexing
  - Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups

```
In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
```

```
In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [130]: data2
Out[130]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

```
In [131]: data2.unstack()
Out[131]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

Stacking filters out missing data by default

```
In [133]: data2.unstack().stack()
Out[133]:
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
```

dropna=False, keeps missing data

```
In [134]: data2.unstack().stack(dropna=False)
Out[134]:
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
```

- Reshaping and Pivoting: Pivoting “Long” to “Wide” Format
  - A common way to store multiple time series in databases and CSV is in so-called long or stacked format
  - Let’s load some example data and do a small amount of time series wrangling and other data cleaning

```
In [139]: data = pd.read_csv('macrodata.csv')
```

```
In [140]: data.head()
Out[140]:
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi    cpi  \
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9  28.98
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7  29.15
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4  29.35
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3  29.37
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5  29.54
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19
```

PeriodIndex combines the year and quarter columns to create a kind of time interval type

```
In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....:                          name='date')
```

```
In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to_timestamp('D', 'end')

In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

- Reshaping and Pivoting: Pivoting “Long” to “Wide” Format
  - After stacking
  - **ldata** looks like:

```
In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

```
In [146]: ldata[:10]
Out[146]:
        date     item     value
0 1959-03-31  realgdp  2710.349
1 1959-03-31     infl     0.000
2 1959-03-31    unemp     5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30     infl     2.340
5 1959-06-30    unemp     5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30     infl     2.740
8 1959-09-30    unemp     5.300
9 1959-12-31  realgdp  2785.204
```

Data is frequently stored this way in relational databases like MySQL

- This is the so-called **long format** for multiple time series, or other observational data with two or more keys (here, date and item) and the value
- Each row in the table represents a single observation (e. g. a sensor sample registration)

- Reshaping and Pivoting: Pivoting “Long” to “Wide” Format
  - In some cases, the data may be more difficult to work with in this format
    - when it is preferable to have a DataFrame containing one column per distinct item value indexed by timestamps in the date column
  - DataFrame’s **pivot** method performs this transformation

```
In [146]: ldata[:10]
Out[146]:
        date     item     value
0 1959-03-31  realgdp  2710.349
1 1959-03-31     infl     0.000
2 1959-03-31    unemp     5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30     infl     2.340
5 1959-06-30    unemp     5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30     infl     2.740
8 1959-09-30    unemp     5.300
9 1959-12-31  realgdp  2785.204
```

```
In [147]: pivoted = ldata.pivot('date', 'item', 'value')

In [148]: pivoted
Out[148]:
item          infl    realgdp  unemp
date
1959-03-31    0.00   2710.349    5.8
1959-06-30    2.34   2778.801    5.1
1959-09-30    2.74   2775.488    5.3
1959-12-31    0.27   2785.204    5.6
1960-03-31    2.31   2847.699    5.2
1960-06-30    0.14   2834.390    5.2
1960-09-30    2.70   2839.022    5.6
1960-12-31    1.21   2802.616    6.3
1961-03-31   -0.40   2819.264    6.8
1961-06-30    1.47   2872.005    7.0
...            ...        ...    ...
2007-06-30    2.75  13203.977    4.5
2007-09-30    3.45  13321.109    4.7
2007-12-31    6.38  13391.249    4.8
2008-03-31    2.82  13366.865    4.9
2008-06-30    8.53  13415.266    5.4
2008-09-30   -3.16  13324.600    6.0
2008-12-31   -8.79  13141.920    6.9
2009-03-31    0.94  12925.410    8.1
2009-06-30    3.37  12901.504    9.2
2009-09-30    3.56  12990.341    9.6
[203 rows x 3 columns]
```

- Reshaping and Pivoting: Pivoting “Long” to “Wide” Format
  - When having two value columns to reshape simultaneously, DataFrame’s **pivot** method performs this transformation

```
In [149]: ldata['value2'] = np.random.randn(len(ldata))

In [150]: ldata[:10]
Out[150]:
        date     item     value    value2
0 1959-03-31  realgdp  2710.349  0.523772
1 1959-03-31     infl     0.000  0.000940
2 1959-03-31    unemp     5.800  1.343810
3 1959-06-30  realgdp  2778.801 -0.713544
4 1959-06-30     infl     2.340 -0.831154
5 1959-06-30    unemp     5.100 -2.370232
6 1959-09-30  realgdp  2775.488 -1.860761
7 1959-09-30     infl     2.740 -0.860757
8 1959-09-30    unemp     5.300  0.560145
9 1959-12-31  realgdp  2785.204 -1.265934
```

```
In [151]: pivoted = ldata.pivot('date', 'item')

In [152]: pivoted[:5]
Out[152]:
            value                          value2
item         infl    realgdp unemp       infl    realgdp     unemp
date
1959-03-31   0.00  2710.349   5.8   0.000940   0.523772  1.343810
1959-06-30   2.34  2778.801   5.1  -0.831154  -0.713544 -2.370232
1959-09-30   2.74  2775.488   5.3  -0.860757  -1.860761  0.560145
1959-12-31   0.27  2785.204   5.6   0.119827  -1.265934 -1.063512
1960-03-31   2.31  2847.699   5.2  -2.359419   0.332883 -0.199543
```

- Reshaping and Pivoting: Pivoting “Wide” to “Long” Format
  - An inverse operation to **pivot** for DataFrames is **pandas.melt**
  - **melt** merges multiple columns into one, producing a DataFrame that is longer than the input

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....:                    'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6],
   .....:                    'C': [7, 8, 9]})

In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

```
In [159]: melted = pd.melt(df, ['key'])

In [160]: melted
Out[160]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

  - The 'key' column may be a group indicator, and the other columns are data values

- Reshaping and Pivoting: Pivoting “Wide” to “Long” Format
  - Using **pivot**, it is possible to reshape back to the original layout

```
In [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped
Out[162]:
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

  - The result of pivot creates an index from the column used as the row labels, so to move the data back into a column, use **reset_index**

```
In [163]: reshaped.reset_index()
Out[163]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

- Reshaping and Pivoting: Pivoting “Wide” to “Long” Format
  - To specify a subset of columns to use as value columns

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

  - Although **pandas.melt** can be used without any group identifiers, too

```
In [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
  variable  value
0        A      1
1        A      2
2        A      3
3        B      4
4        B      5
5        B      6
6        C      7
7        C      8
8        C      9
```