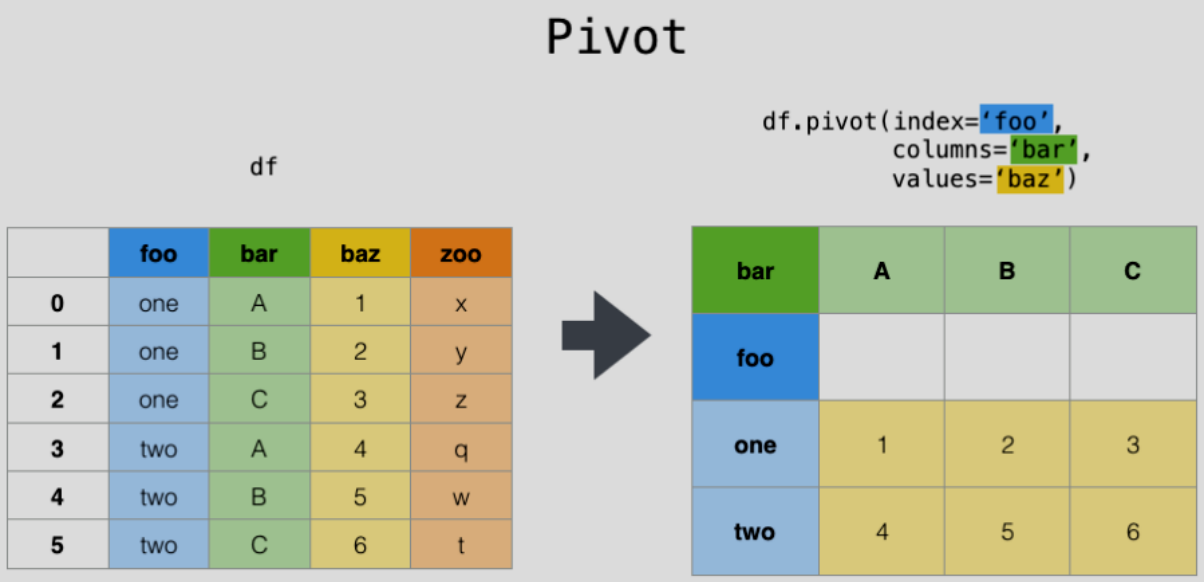
Reshaping and Pivot tables

In Pandas data reshaping means the transformation of the structure of a table or vector (i.e. DataFrame or Series) to make it suitable for further analysis.

The pivot function is used to create a new derived table out of a given one. Pivot takes 3 arguments with the following names: index, columns, and values. As a value for each of these parameters you need to specify a column name in the original table. Then the pivot function will create a new table, whose row and column indices are the unique values of the respective parameters. The cell values of the new table are taken from column given as the values parameter.

Reshaping by pivoting DataFrame Objects:



Data is often stored in “stacked” or “record” format. Let’s take a DataFrame as Example and test every function. Let the dataset consists of date,variable and values as columns. Let’s say:

```
          date variable     value
0   2000-01-03        A  0.469112
1   2000-01-04        A -0.282863
2   2000-01-05        A -1.509059
3   2000-01-03        B -1.135632
4   2000-01-04        B  1.212112
5   2000-01-05        B -0.173215
6   2000-01-03        C  0.119209
7   2000-01-04        C -1.044236
8   2000-01-05        C -0.861849
9   2000-01-03        D -2.104569
10  2000-01-04        D -0.494929
11  2000-01-05        D  1.071804
```

To Select a specific variable “A” from the data we can filter the specific variable using filters in DataFrame but if we want to work on a time series data

we need a better representation of the dataset so that we can understand better and work on it where the columns are the unique variables and an index of dates identifies individual observations. To reshape the data into this form, we use the DataFrame.pivot() method. Then the DataFrame will transform like:

```
variable           A         B         C         D
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804
```

If the values argument is omitted, and the input DataFrame has more than one column of values which are not used as column or index inputs to pivot(), then the resulting "pivoted" DataFrame will have hierarchical columns whose topmost level indicates the respective value column:

```
              value                          ...    value2
variable          A         B         C  ...         B         C         D
date                                       ...
2000-01-03  0.469112 -1.135632  0.119209  ... -2.271265  0.238417 -4.209138
2000-01-04 -0.282863  1.212112 -1.044236  ...  2.424224 -2.088472 -0.989859
2000-01-05 -1.509059 -0.173215 -0.861849  ... -0.346429 -1.723698  2.143608
```

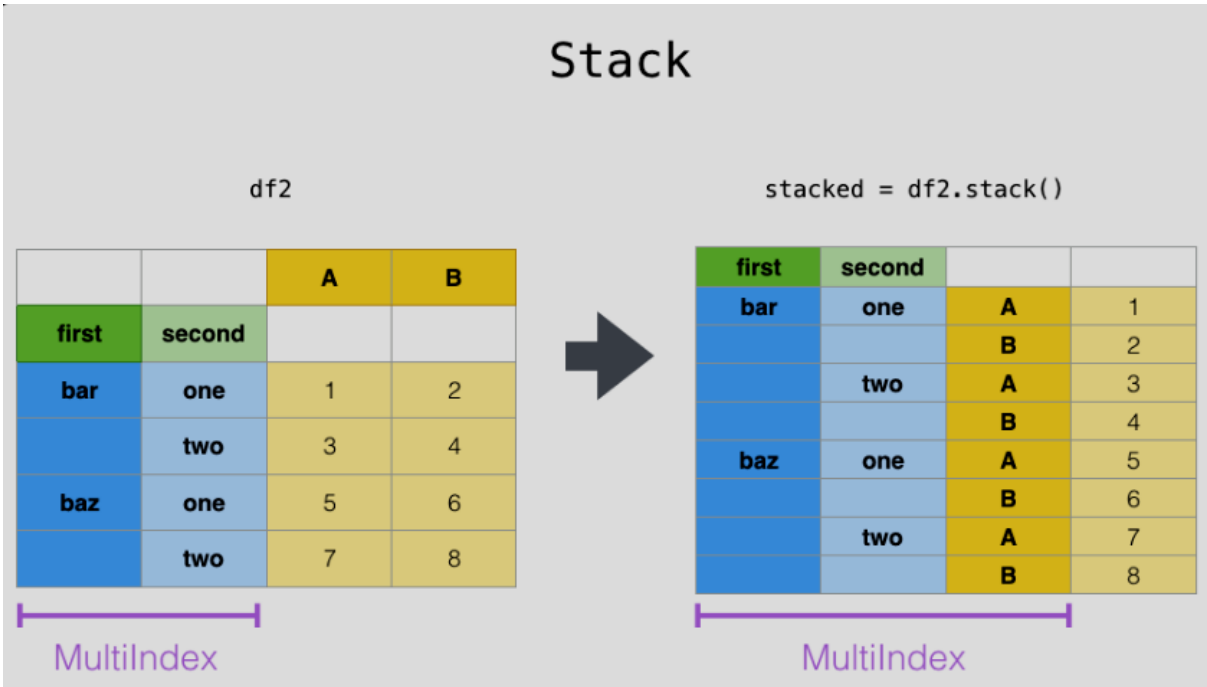
We can select any subsets we want from the pivot table using pivoted["column name"]:

```
variable           A         B         C         D
date
2000-01-03  0.938225 -2.271265  0.238417 -4.209138
2000-01-04 -0.565727  2.424224 -2.088472 -0.989859
2000-01-05 -3.018117 -0.346429 -1.723698  2.143608
```

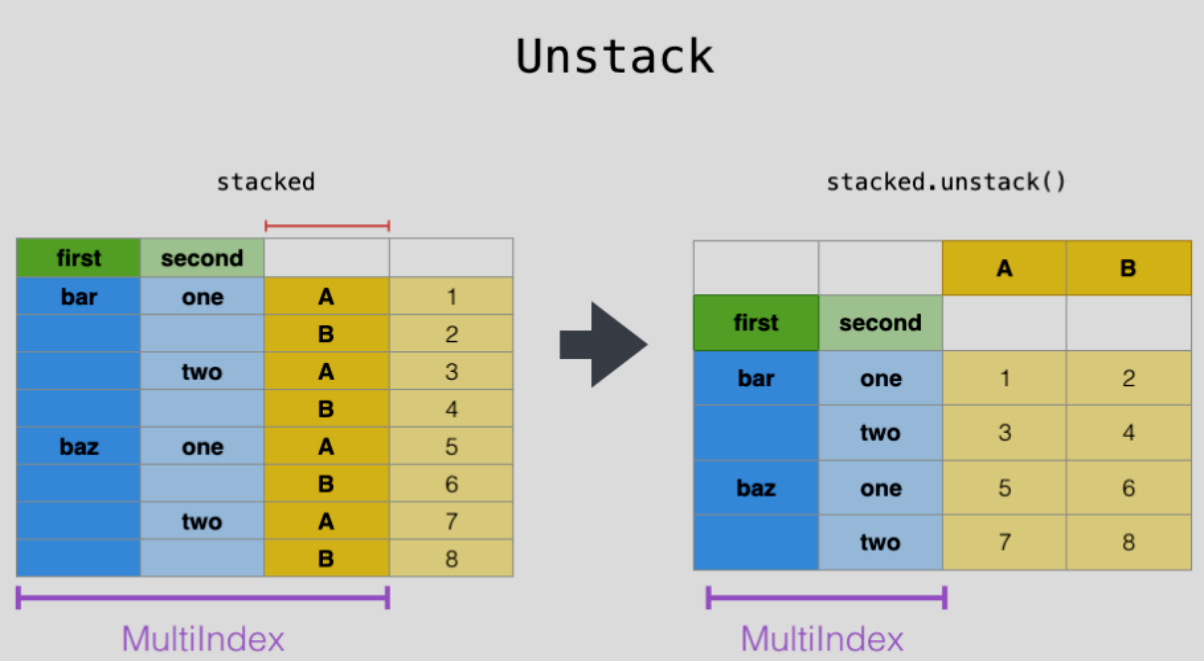
Reshaping by stacking and unstacking:

It is similar to the pivot() method are the related to stack() and unstack() methods available on series and DataFrame. The methods are designed to work together with MultiIndex objects. Let's take a look on how these methods work:

- Stack(): "pivot" a level of the (possibly hierarchical) column labels, returning a DataFrame with an index with a new inner-most level of row labels.



- Unstack(): (inverse operation of stack()) "pivot" a level of the (possibly hierarchical) row index to the column axis, producing a reshaped DataFrame with a new inner-most level of column labels.



The clearest way to explain is by example. Let's take a prior example data set from the hierarchical indexing section. Adding tuple as a zip with bar and baz as the new values using multiIndex we set first and second as 2 index's/ Let's say the DataFrame be like:

```
                     A         B
first second
bar   one      0.721555 -0.706771
      two     -1.039575  0.271860
baz   one     -0.424972  0.567020
      two      0.276232 -1.087401
```

The stack() function "compresses" a level in the DataFrame columns to produce either:

- A Series, in the case of a simple column Index.
- A DataFrame, in the case of a MultiIndex in the columns.

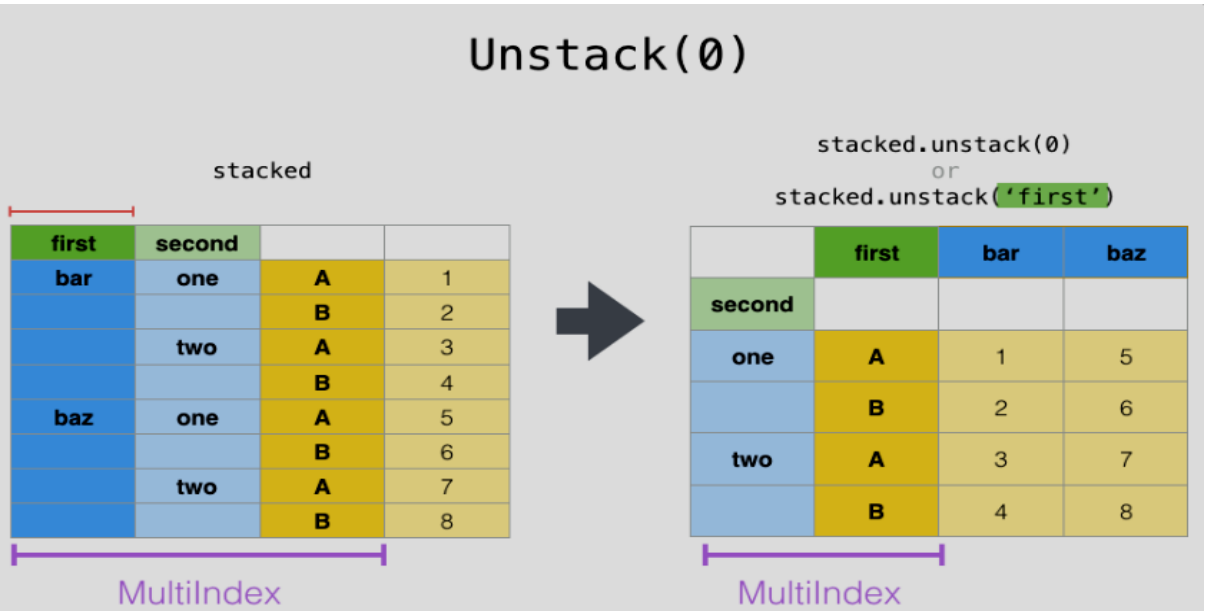
If the columns have a MultiIndex, you can choose which level to stack. The stacked level becomes the new lowest level in a MultiIndex on the column. Then the stacked DataFrame look like:

```
first  second
bar    one     A    0.721555
               B   -0.706771
       two     A   -1.039575
               B    0.271860
baz    one     A   -0.424972
               B    0.567020
       two     A    0.276232
               B   -1.087401
```

With a "stacked" DataFrame or Series (having a MultiIndex as the index), the inverse operation of stack() is unstack(), which by default unstacks the last level and it becomes as:

```
                     A         B
first second
bar   one      0.721555 -0.706771
      two     -1.039575  0.271860
baz   one     -0.424972  0.567020
      two      0.276232 -1.087401
```

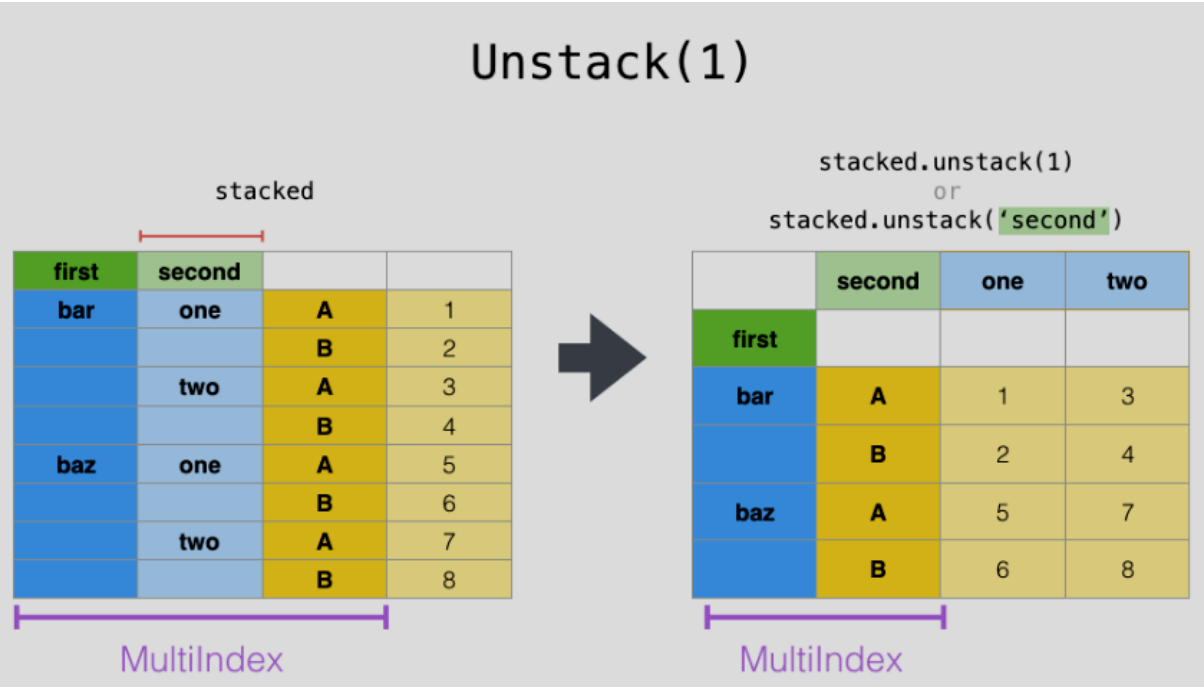
can unstack the stacked DataFrame in different levels like unstack(0)



which returns the first as the hierarchal when applied to our stacked DataFrame and looks like:

```
first          bar       baz
second
one    A  0.721555 -0.424972
       B -0.706771  0.567020
two    A -1.039575  0.276232
       B  0.271860 -1.087401
```

Similarly, unstack(1) which returns second as the hierarchal column:



When we apply the unstack(1) to our DataFrame it returns:

```
second        one       two
first
bar   A  0.721555 -1.039575
      B -0.706771  0.271860
baz   A -0.424972  0.276232
      B  0.567020 -1.087401
```

Multiple levels:

By passing a list of levels we can also stack or unstack more than one level at a time, in which case the end result is as if each level in the list were processed individually Like ‘names=["exp", "animal", "hair_length"]’ which looks like when applied to a DataFrame:

```
exp                A         B         A         B
animal           cat       cat       dog       dog
hair_length     long      long     short     short
0            1.075770 -0.109050  1.643563 -1.469388
1            0.357021 -0.674600 -1.776904 -0.968914
2           -1.294524  0.413738  0.276662 -0.472035
3           -0.013960 -0.362543 -0.006154 -0.923061
```

When we stack this by using the level as animal and hair_length. We get the DataFrame as:

```
exp                           A         B
  animal hair_length
0 cat    long          1.075770 -0.109050
  dog    short         1.643563 -1.469388
1 cat    long          0.357021 -0.674600
  dog    short        -1.776904 -0.968914
2 cat    long         -1.294524  0.413738
  dog    short         0.276662 -0.472035
3 cat    long         -0.013960 -0.362543
  dog    short        -0.006154 -0.923061
```

Note: The list of the levels which are given to stack can only be one at a time like either numbers or names at a time not a mixture of both.

Missing Data:

Missing data functions in pandas are intelligent about handling missing data and do not expect each subgroup within the hierarchical index to have the same set of labels. They also can handle the index being unsorted (but you can make it sorted by calling sort_index(), of course). Here is a more complex example:

Let’s take MultiIndex from tuples and give them names and another MultiIndex from product with names. DataFrame looks like:

```
exp                A         B                   A
animal           cat       dog       cat       dog
first second
bar   one     0.895717  0.805244 -1.206412  2.565646
      two     1.431256  1.340309 -1.170299 -0.226169
baz   one     0.410835  0.813850  0.132003 -0.827317
foo   one    -1.413681  1.607920  1.024180  0.569605
      two     0.875906 -2.211372  0.974466 -2.006747
qux   two    -1.226825  0.769804 -1.281247 -0.727707
```

As mentioned above, stack() can be called with a level argument to select which level in the columns to stack. If we pass a command with .stack("exp") then the DataFrame returns:

```
animal                  cat       dog
first second exp
bar   one    A     0.895717  2.565646
             B    -1.206412  0.805244
      two    A     1.431256 -0.226169
             B    -1.170299  1.340309
baz   one    A     0.410835 -0.827317
             B     0.132003  0.813850
foo   one    A    -1.413681  0.569605
             B     1.024180  1.607920
      two    A     0.875906 -2.006747
             B     0.974466 -2.211372
qux   two    A    -1.226825 -0.727707
             B    -1.281247  0.769804
```

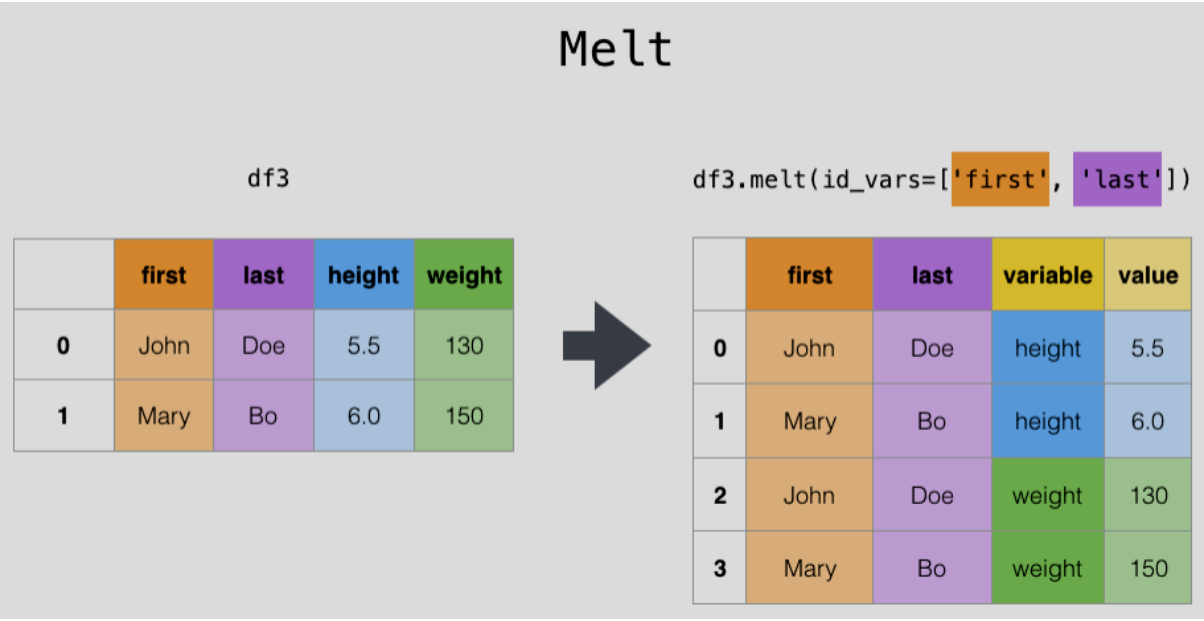
Unstacking can result in missing values if subgroups do not have the same set of labels. By default, missing values will be replaced with the default fill value for that data type, NaN for float, NaT for datetimelike, etc. For integer types, by default data will converted to float and missing values will be set to NaN. Alternatively, unstack takes an optional fill_value argument, for specifying the value of missing data.

With a MultiIndex

Unstacking when the columns are a MultiIndex is also careful about doing the right thing:

```
exp            A                   B  ...                   A
animal       cat                 dog  ...       cat       dog
first        bar       baz       bar  ...       baz       bar       baz
second                               ...
one     0.895717  0.410835  0.805244  ...  0.132003  2.565646 -0.827317
two     1.431256       NaN  1.340309  ...       NaN -0.226169       NaN
```

Reshaping by melt



The top-level melt() function and the corresponding DataFrame.melt() are useful to massage a DataFrame into a format where one or more columns are *identifier variables*, while all other columns, considered *measured variables*, are “unpivoted” to the row axis, leaving just two non-identifier columns, “variable” and “value”. The names of those columns can be customized by supplying the var_name and value_name parameters.

For instance, Let’s take a DataFrame consists of a person details which includes first name, last name, height and weight. DataFrame look like:

```
  first last  height  weight
0  John  Doe     5.5     130
1  Mary   Bo     6.0     150
```

After using melt func with specific ids like first and last we get the DataFrame looks like:

```
  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

We can even add new variable name as quantity so that we can see whole DataFrame like this:

```
  first last quantity  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

Combining with stats and GroupBy

It should be no shock that combining pivot() / stack() / unstack() with GroupBy and the basic Series and DataFrame statistical functions can produce some very expressive and fast data manipulations.

1: Unstacking with mean as the stat factor,

```
animal             cat       dog
first second
bar   one    -0.155347  1.685445
      two     0.130479  0.557070
baz   one     0.271419 -0.006733
      two     0.526830 -1.312207
foo   one    -0.194750  1.088763
      two     0.925186 -2.109060
qux   one     0.067976 -0.648927
      two    -1.254036  0.021048
```

We can represent these with many other ways by using GroupBy functions assigning different levels and axis with values 1,

```
animal             cat       dog
first second
bar   one    -0.155347  1.685445
      two     0.130479  0.557070
baz   one     0.271419 -0.006733
      two     0.526830 -1.312207
foo   one    -0.194750  1.088763
      two     0.925186 -2.109060
qux   one     0.067976 -0.648927
      two    -1.254036  0.021048
```

Pivot tables

While pivot() provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides pivot_table() for pivoting with aggregation of numeric data.

The function pivot_table() can be used to create spreadsheet-style pivot tables. See the cookbook for some advanced strategies.

It takes a number of arguments:

- data: a DataFrame object.
- values: a column or a list of columns to aggregate.
- index: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
- columns: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
- aggfunc: function to use for aggregation, defaulting to numpy.mean.

Consider a data set like this:

```
        A  B    C         D         E          F
0     one  A  foo  0.341734 -0.317441 2013-01-01
1     one  B  foo  0.959726 -1.236269 2013-02-01
2     two  C  foo -1.110336  0.896171 2013-03-01
3   three  A  bar -0.619976 -0.487602 2013-04-01
4     one  B  bar  0.149748 -0.082240 2013-05-01
..    ... ..  ...       ...       ...        ...
19  three  B  foo  0.690579 -2.213588 2013-08-15
20    one  C  foo  0.995761  1.063327 2013-09-15
21    one  A  bar  2.396780  1.266143 2013-10-15
22    two  B  bar  0.014871  0.299368 2013-11-15
23  three  C  bar  3.357427 -0.863838 2013-12-15
```

From this data we can produce pivot tables very easily:

Used Command:

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

```
C             bar       foo
A     B
one   A  1.120915 -0.514058
      B -0.338421  0.002759
      C -0.538846  0.699535
three A -1.181568       NaN
      B       NaN  0.433512
      C  0.588783       NaN
two   A       NaN  1.000985
      B  0.158248       NaN
      C       NaN  0.176180
```

The result object is a DataFrame having potentially hierarchical indexes on the rows and columns. If the values column name is not given, the pivot table will include all of the data in an additional level of hierarchy in the columns:

```
               D                   E
C            bar       foo       bar       foo
A     B
one   A  1.120915 -0.514058  1.393057 -0.021605
      B -0.338421  0.002759  0.684140 -0.551692
      C -0.538846  0.699535 -0.988442  0.747859
three A -1.181568       NaN  0.961289       NaN
      B       NaN  0.433512       NaN -1.064372
      C  0.588783       NaN -0.131830       NaN
two   A       NaN  1.000985       NaN  0.064245
      B  0.158248       NaN -0.097147       NaN
      C       NaN  0.176180       NaN  0.436241
```

Also, you can use Grouper for index and columns keywords. For detail of Grouper, see Grouping with a Grouper specification.

```
C               bar       foo
F
2013-01-31      NaN -0.514058
2013-02-28      NaN  0.002759
2013-03-31      NaN  0.176180
2013-04-30 -1.181568       NaN
2013-05-31 -0.338421       NaN
2013-06-30 -0.538846       NaN
2013-07-31      NaN  1.000985
2013-08-31      NaN  0.433512
2013-09-30      NaN  0.699535
2013-10-31  1.120915       NaN
2013-11-30  0.158248       NaN
2013-12-31  0.588783       NaN
```

You can render a nice output of the table omitting the missing values by calling to_string() if you wish:

```
                D                   E
C             bar       foo       bar       foo
A     B
one   A  1.120915 -0.514058  1.393057 -0.021605
      B -0.338421  0.002759  0.684140 -0.551692
      C -0.538846  0.699535 -0.988442  0.747859
three A -1.181568            0.961289
      B            0.433512           -1.064372
      C  0.588783           -0.131830
two   A            1.000985            0.064245
      B  0.158248           -0.097147
      C            0.176180            0.436241
```

Note that pivot_table() is also available as an instance method on DataFrame, i.e. DataFrame.pivot_table().

Adding Margins:

If we pass margins=True to **pivot_table()**, special All columns and rows will be added with partial group aggregates across the categories on the rows and columns:

```
                D                             E
C             bar       foo       All       bar       foo       All
A     B
one   A  1.804346  1.210272  1.569879  0.179483  0.418374  0.858005
      B  0.690376  1.353355  0.898998  1.083825  0.968138  1.101401
      C  0.273641  0.418926  0.771139  1.689271  0.446140  1.422136
three A  0.794212       NaN  0.794212  2.049040       NaN  2.049040
      B       NaN  0.363548  0.363548       NaN  1.625237  1.625237
      C  3.915454       NaN  3.915454  1.035215       NaN  1.035215
two   A       NaN  0.442998  0.442998       NaN  0.447104  0.447104
      B  0.202765       NaN  0.202765  0.560757       NaN  0.560757
      C       NaN  1.819408  1.819408       NaN  0.650439  0.650439
All      1.556686  0.952552  1.246608  1.250924  0.899904  1.059389
```

Additionally, we can call **DataFrame.stack()** to display a pivoted DataFrame as having a multi-level index:

```
                     D         E
A   B C
one A All  1.569879  0.858005
      bar  1.804346  0.179483
      foo  1.210272  0.418374
    B All  0.898998  1.101401
      bar  0.690376  1.083825
...              ...       ...
two C All  1.819408  0.650439
      foo  1.819408  0.650439
All   All  1.246608  1.059389
      bar  1.556686  1.250924
      foo  0.952552  0.899904
```

Cross Tabulations:

Use crosstab() to compute a cross-tabulation of two (or more) factors. By default crosstab() computes a frequency table of the factors unless an array of values and an aggregation function are passed.

It takes a number of arguments

- index: array-like, values to group by in the rows.
- columns: array-like, values to group by in the columns.
- values: array-like, optional, array of values to aggregate according to the factors.
- aggfunc: function, optional, If no values array is passed, computes a frequency table.
- rownames: sequence, default None, must match number of row arrays passed.
- colnames: sequence, default None, if passed, must match number of column arrays passed.
- margins: boolean, default False, Add row/column margins (subtotals)
- normalize: boolean, {'all', 'index', 'columns'}, or {0,1}, default False. Normalize by dividing all values by the sum of values.

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified

For example:

Let's take a DataFrame with Random data,

b	one		two	
c	dull	shiny	dull	shiny
a				
bar	1	0	0	1
foo	2	1	1	0

If **crosstab()** receives only two Series, it will provide a frequency table.

	A	B	C
0	1	3	1.0
1	2	3	1.0
2	2	4	NaN
3	2	4	1.0
4	2	4	1.0

After Using Crosstab function we get,

B	3	4
A		
1	1	0
2	1	3

Note: **crosstab()** can also be implemented to **Categorical** data.

Normalization

Frequency tables can also be normalized to show percentages rather than counts using the normalize argument:

If we pass normalize='True' we get,

```
B    3    4
A
1  0.2  0.0
2  0.2  0.6
```

- normalize can also normalize values within each row or within each column by passing normalize

crosstab() can also be passed a third **Series** and an aggregation function (aggfunc) that will be applied to the values of the third **Series** within each group defined by the first two **Series**:

```
Ex: pd.crosstab(df["A"], df["B"], values=df["C"], aggfunc=np.sum)
Out[91]:
```

```
B    3    4
A
1  1.0  NaN
2  1.0  2.0
```

Tiling:

The cut() function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

Example: Let's take an array of few integers and apply cut() function. We get output as,

```
[(9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95,
26.667], (9.95, 26.667], (26.667, 43.333], (43.333, 60.0], (43.333, 60.0]]
Categories (3, interval[float64, right]): [(9.95, 26.667] < (26.667,
43.333] < (43.333, 60.0]]
```

If the bins keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges. We get:

```
[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35,
70], (35, 70]]
Categories (3, interval[int64, right]): [(0, 18] < (18, 35] < (35, 70]]
```

Computing Indicator / dummy variables:

To convert a categorical variable into a “dummy” or “indicator” **DataFrame**, for example a column in a **DataFrame** (a **Series**) which has k distinct values, can derive a **DataFrame** containing k columns of 1s and 0s using **get_dummies()**:

Example:

Let’s take a DataFrame with some keys and values in in with certain range and apply the function get_dummies. We get,

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

This function is often used along with discretization functions like **cut()**:

Suppose let the array be:

```
array([ 0.4082, -1.0481, -0.0257, -0.9884,  0.0941,  1.2627,  1.29  ,
        0.0824, -0.0558,  0.5366])
```

Bins: [0, 0.2, 0.4, 0.6, 0.8, 1]

Such that if we apply get_dummies we get,

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	1	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	1	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	1	0	0	0	0
8	0	0	0	0	0
9	0	0	1	0	0

To convert a “dummy” or “indicator” DataFrame, into a categorical DataFrame, for example k columns of a DataFrame containing 1s and 0s can derive a DataFrame which has k distinct values using **from_dummies()**:

Let the DataFrame be:

	prefix_a	prefix_b
0	0	1
1	1	0
2	0	1

If we apply from_dummies for the DataFrame we get,

```
  prefix
0      b
1      a
2      b
```

Dummy coded data only requires k - 1 categories to be included, in this case the k th category is the default category, implied by not being assigned any of the other k - 1 categories, can be passed via default_category.

```
Df:          prefix_a              Out:          prefix
          0         0                          0      b
          1         1                          1      a
          2         0                          2      b
```

Factorizing values:

To encode 1-d values as an enumerated type use **factorize()**:

Let's take a DataFrame:

```
0       A
1       A
2     NaN
3       B
4    3.14
5     inf
```

If we pass uniques we get,

```
Index(['A', 'B', 3.14, inf], dtype='object')
```

Note: **factorize()** is similar to numpy.unique, but differs in its handling of NaN

Exploding a list-like column:

Sometimes the values in a column are list-like.

Let's take a DataFrame with some keys and values such that the DataFrame Looks like:

```
     keys            values
0  panda1    [eats, shoots]
1  panda2  [shoots, leaves]
2  panda3    [eats, leaves]
```

We can 'explode' the values column, transforming each list-like to a separate row, by using **explode()**. This will replicate the index values from the original row:

Let's apply the function: `df["values"].explode()`

```
0      eats
0    shoots
1    shoots
1    leaves
2      eats
2    leaves
```

`Series.explode()` will replace empty lists with np.nan and preserve scalar entries. The dtype of the resulting **`Series`** is always object.

Let's take a series DataFrame:

```
0    [1, 2, 3]
1          foo
2           []
3       [a, b]
```

And lets explode it,

```
0      1
0      2
0      3
1    foo
2    NaN
3      a
3      b
```

Here is a typical usecase. You have comma separated strings in a column and want to expand this.

Let's take another Df:

```
    var1  var2
0  a,b,c     1
1  d,e,f     2
```

And use this `df.assign(var1=df.var1.str.split(",")).explode("var1")`

We get,

```
  var1  var2
0    a     1
0    b     1
0    c     1
1    d     2
1    e     2
1    f     2
```