Long and wide data format

There are several ways to store the same data.

Long format

In [1]:

```
import pandas as pd
df_long = pd.DataFrame({'Animal': ('cat', 'cat', 'dog', 'dog', 'cow', 'cow'),
                        'Feature': ('Age', 'Mass', 'Age', 'Mass', 'Age', 'Mass'),
                        'Value': (11, 5, 8, 17, 4, 650)})
```

In [2]:
```
df_long
```

Out[2]:

	Animal	Feature	Value
0	cat	Age	11
1	cat	Mass	5
2	dog	Age	8
3	dog	Mass	17
4	cow	Age	4
5	cow	Mass	650

Above, some data is stored using the **long format** relatively to column `Animal`. This means several rows have the sale 'Animal' value.

The **long format** :

- makes DataFrame having few comumns but many rows
- makes it difficult to work on specific values. For instance, how to perform calculation on the mass of all animals?

From long to wide format

Thus, let's transform the data to have it in **large format**. This is done using `pivot_table`.

In [3]:
```
df_wide = df_long.pivot_table(index='Animal', columns='Feature', values='Value')
df_wide
```

Out[3]:

Feature	Age	Mass
Animal		
cat	11.0	5.0
cow	4.0	650.0
dog	8.0	17.0

Above, `df_wide` has as many columns as there are different elements in the `Feature` column of `df_long`. The name 'Feature' is given to the index along axis 1, i.e. the columns.

In [4]:
```
df_wide.columns.name
```

Out[4]:
```
'Feature'
```

Processing math: 100%

From wide to long format

Conversely, the `melt` function makes it possible to transform data from a wide to a long format:

In [5]:

```
df_wide = df_wide.reset_index()
df_wide.melt(id_vars='Animal', value_vars=['Age', 'Mass'],
             var_name='Feature', value_name='Value')
```

Out[5]:

	Animal	Feature	Value
0	cat	Age	11.0
1	cow	Age	4.0
2	dog	Age	8.0
3	cat	Mass	5.0
4	cow	Mass	650.0
5	dog	Mass	17.0

Processing math: 100%

Advanced

In the previous example `df_long` has only one value `Value` for each (`Animal`, `Feature`) pair. Whenever it's not the case, a `aggfunc` must be specified when going from long to wide format.

Processing math: 100%

Another DataFrame definition:

In [6]:
```
df_long = pd.DataFrame({'Animal': ('cat', 'cat', 'cat', 'dog', 'dog', 'dog', 'cow', '
                        'Feature': ('Age', 'Mass', 'Mass', 'Age', 'Mass', 'Mass', 'Age', '
                        'Value': (11, 5, 9, 8, 17, 11, 4, 650)})
df_long
```

Out[6]:

	Animal	Feature	Value
0	cat	Age	11
1	cat	Mass	5
2	cat	Mass	9
3	dog	Age	8
4	dog	Mass	17
5	dog	Mass	11
6	cow	Age	4
7	cow	Mass	650

Processing math: 100%

`df_long` has now 2 masses for the cat and the dog. Let's define `aggfunc` :

In [7]:
```
df_long.pivot_table(index='Animal', columns='Feature', values='Value', aggfunc='mean'
```

Out[7]:

Feature	Age	Mass
Animal		
cat	11.0	7.0
cow	4.0	650.0
dog	8.0	14.0

In [8]:
```
df_long.pivot_table(index='Animal', columns='Feature', values='Value', aggfunc=list)
```

Out[8]:

Feature	Age	Mass
Animal		
cat	[11]	[5, 9]
cow	[4]	[650]
dog	[8]	[17, 11]

Index swapping

A DataFrame has two indexes:

- along rows (axis 0): can be accessed using `.index`
- along columns (axis 1): can be accessed using `.columns`

The `stack` method can append the column index to rows. `unstack` do the opposite.

Processing math: 100%

`stack`

In [9]:
```
df_wide
```

Out[9]:

Feature	Animal	Age	Mass
0	cat	11.0	5.0
1	cow	4.0	650.0
2	dog	8.0	17.0

In [10]:
```
df_wide_stacked = df_wide.stack()
df_wide_stacked
```

Out[10]:
```
   Feature
0  Animal       cat
   Age         11.0
   Mass         5.0
1  Animal       cow
   Age          4.0
   Mass       650.0
2  Animal       dog
   Age          8.0
   Mass        17.0
dtype: object
```

Processing math: 100%

Since there was only one level of columns, the call to `stack` returns a `Serie`.

In [11]:
```
type(df_wide_stacked)
```

Out[11]:
```
pandas.core.series.Series
```

Processing math: 100%

And since there already was an index, there are now 2 of them (multi index):

In [12]:
```
df_wide_stacked.index
```

Out[12]:
```
MultiIndex([(0, 'Animal'),
            (0,    'Age'),
            (0,   'Mass'),
            (1, 'Animal'),
            (1,    'Age'),
            (1,   'Mass'),
            (2, 'Animal'),
            (2,    'Age'),
            (2,   'Mass')],
           names=[None, 'Feature'])
```

Processing math: 100%

Multi index can be accessed this way:

In [13]:
```
df_wide_stacked.loc[(1, 'Age')]
```

Out[13]:
```
4.0
```

Processing math: 100%

`unstack`

Using `unstack`, the row index becomes a columns index. Thus, the multi index is now at the column level and there is no more index at the row level:

In [14]:
```
df_wide
```

Out[14]:

Feature	Animal	Age	Mass
0	cat	11.0	5.0
1	cow	4.0	650.0
2	dog	8.0	17.0

In [15]:
```
df_wide_unstacked = df_wide.unstack()
df_wide_unstacked
```

Out[15]:
```
Feature
Animal   0      cat
         1      cow
         2      dog
Age      0     11.0
         1      4.0
         2      8.0
Mass     0      5.0
         1    650.0
         2     17.0
dtype: object
```

Processing math: 100%

Use case

`stack` and `unstack` are very powerful whenever the DataFrame has an index (rows or columns) with more than one level.

DataFrame definition

In [16]:

```
import numpy as np
index_data_rows = [[1, 1, 2, 2, 3], ['x', 'y', 'x', 'y', 'z']]
index_rows = pd.MultiIndex.from_arrays(index_data_rows,
                                       names=('level_0_rows', 'level_1_rows'))

index_data_cols = [['a', 'a', 'b'], ['A', 'B', 'B']]
index_cols = pd.MultiIndex.from_arrays(index_data_cols, names=('level_0_cols', 'level
df = pd.DataFrame(data=np.arange(15).reshape((5, 3)),
                  columns=index_cols,
                  index=index_rows)
```

Processing math: 100%

In [17]:
```
df
```

Out[17]:

	level_0_cols	a	a	b
	level_1_cols	A	B	B
level_0_rows	level_1_rows			
1	x	0	1	2
	y	3	4	5
2	x	6	7	8
	y	9	10	11
3	z	12	13	14

Processing math: 100%

In [18]:
```
df.index
```

Out[18]:
```
MultiIndex([(1, 'x'),
            (1, 'y'),
            (2, 'x'),
            (2, 'y'),
            (3, 'z')],
           names=['level_0_rows', 'level_1_rows'])
```

In [19]:
```
df.columns
```

Out[19]:
```
MultiIndex([('a', 'A'),
            ('a', 'B'),
            ('b', 'B')],
           names=['level_0_cols', 'level_1_cols'])
```

Processing math: 100%

`unstack`

In [20]:
```
unstacked = df.unstack()
```

In [21]:
```
unstacked
```

Out[21]:

level_0_cols			a						b
level_1_cols			A			B			B
level_1_rows	x	y	z	x	y	z	x	y	z
level_0_rows									
1	0.0	3.0	NaN	1.0	4.0	NaN	2.0	5.0	NaN
2	6.0	9.0	NaN	7.0	10.0	NaN	8.0	11.0	NaN
3	NaN	NaN	12.0	NaN	NaN	13.0	NaN	NaN	14.0

Processing math: 100%

In [22]:
```
unstacked.loc[2, ('a', 'B', 'y')]
```
Out[22]:
```
10.0
```
In [23]:
```
unstacked.index
```
Out[23]:
```
Index([1, 2, 3], dtype='int64', name='level_0_rows')
```
In [24]:
```
unstacked.columns
```
Out[24]:
```
MultiIndex([('a', 'A', 'x'),
            ('a', 'A', 'y'),
            ('a', 'A', 'z'),
            ('a', 'B', 'x'),
            ('a', 'B', 'y'),
            ('a', 'B', 'z'),
            ('b', 'B', 'x'),
            ('b', 'B', 'y'),
            ('b', 'B', 'z')],
           names=['level_0_cols', 'level_1_cols', 'level_1_rows'])
```

`stack`

In [25]:
```python
stacked = df.stack()
```

```
/tmp/ipykernel_29263/924736501.py:1: FutureWarning: The previous implementat
ion of stack is deprecated and will be removed in a future version of panda
s. See the What's New notes for pandas 2.1.0 for details. Specify future_sta
ck=True to adopt the new implementation and silence this warning.
  stacked = df.stack()
```

In [26]:
```python
stacked
```

Out[26]:

		level_0_cols	**a**	**b**
level_0_rows	**level_1_rows**	**level_1_cols**		
1	**x**	**A**	0	NaN
		B	1	2.0
	y	**A**	3	NaN
		B	4	5.0
2	**x**	**A**	6	NaN
		B	7	8.0
	y	**A**	9	NaN
		B	10	11.0
3	**z**	**A**	12	NaN
		B	13	14.0

Processing math: 100%

In [27]:
```
stacked.index
```

Out[27]:
```
MultiIndex([(1, 'x', 'A'),
            (1, 'x', 'B'),
            (1, 'y', 'A'),
            (1, 'y', 'B'),
            (2, 'x', 'A'),
            (2, 'x', 'B'),
            (2, 'y', 'A'),
            (2, 'y', 'B'),
            (3, 'z', 'A'),
            (3, 'z', 'B')],
           names=['level_0_rows', 'level_1_rows', 'level_1_cols'])
```

In [28]:
```
stacked.columns
```

Out[28]:
```
Index(['a', 'b'], dtype='object', name='level_0_cols')
```

Processing math: 100%

Grouping data

When dealing with multi dimensional data, you may need to extract global trendlines regarding some specific attributes. This can be done using `groupby`.

In [29]:
```
df_long
```

Out[29]:

	Animal	Feature	Value
0	cat	Age	11
1	cat	Mass	5
2	cat	Mass	9
3	dog	Age	8
4	dog	Mass	17
5	dog	Mass	11
6	cow	Age	4
7	cow	Mass	650

Unique function

Below, let's compute the average of values 'Value' for every pair (`Animal` , `Feature`).

In [30]:
```
df_long.groupby(by=['Animal', 'Feature'])['Value'].mean()
```

Out[30]:
```
Animal  Feature
cat     Age         11.0
        Mass         7.0
cow     Age          4.0
        Mass       650.0
dog     Age          8.0
        Mass        14.0
Name: Value, dtype: float64
```

note: in this particular case, the result is very similar to what would be returned by `melt` .

Processing math: 100%

If several columns exist, the agregate is done everywhere:

In [31]:
```
df_long['Other value'] = range(10, 18)
df_long
```

Out[31]:

	Animal	Feature	Value	Other value
0	cat	Age	11	10
1	cat	Mass	5	11
2	cat	Mass	9	12
3	dog	Age	8	13
4	dog	Mass	17	14
5	dog	Mass	11	15
6	cow	Age	4	16
7	cow	Mass	650	17

In [32]:

```
df_long.groupby(by=['Animal', 'Feature']).mean()
```

Out[32]:

Animal	Feature	Value	Other value
cat	Age	11.0	10.0
	Mass	7.0	11.5
cow	Age	4.0	16.0
	Mass	650.0	17.0
dog	Age	8.0	13.0
	Mass	14.0	14.5

Processing math: 100%

Multiple functions

But one can specify a different aggregate function depending on the column. This is done passing a dictionary to `agg`:

In [33]:
```
df_long.groupby(by=['Animal', 'Feature']).agg({'Value': 'mean', 'Other value': list})
```

Out[33]:

		Value	Other value
Animal	**Feature**		
cat	**Age**	11.0	[10]
	Mass	7.0	[11, 12]
cow	**Age**	4.0	[16]
	Mass	650.0	[17]
dog	**Age**	8.0	[13]
	Mass	14.0	[14, 15]

Processing math: 100%

Iterating over groups

Without aggregating, one can **iterate over groups**.

In [34]:

```
groupby_object = df_long.groupby(by=['Animal', 'Feature'])
```

In [35]:

```
for tuple_, dataframe in groupby_object:
    print(tuple_)
    print(dataframe, end='\n\n')
    if tuple_==('cow', 'Mass'):
        break # stop displaying values
```

```
('cat', 'Age')
  Animal Feature  Value  Other value
0    cat     Age     11           10

('cat', 'Mass')
  Animal Feature  Value  Other value
1    cat    Mass      5           11
2    cat    Mass      9           12

('cow', 'Age')
  Animal Feature  Value  Other value
6    cow     Age      4           16

('cow', 'Mass')
  Animal Feature  Value  Other value
7    cow    Mass    650           17
```

Merging data

Case study

Merging data is needed to work on a unified instance that contains all the relevant information. For instance, here are some datasets having similar features:

In [36]:
```
df1 = pd.DataFrame({'Name': ('Laura', 'Bob', 'Sarah', 'Li'),
                    'Age': (45, 15, 41, 23),
                    'Address': ('Annecy', 'Turin', 'Annecy', 'Chambéry')})
df2 = pd.DataFrame({'Name': ('Sarah', 'Li', 'Pierre', 'David'),
                    'Age': (41, 23, 26, 45),
                    'Address': ('Annecy', 'Paris', 'Geneva', 'Annecy')})
```

Processing math: 100%

In [37]:
```
df1
```

Out[37]:

	Name	Age	Address
0	Laura	45	Annecy
1	Bob	15	Turin
2	Sarah	41	Annecy
3	Li	23	Chambéry

In [38]:
```
df2
```

Out[38]:

	Name	Age	Address
0	Sarah	41	Annecy
1	Li	23	Paris
2	Pierre	26	Geneva
3	David	45	Annecy

Note that:

- A row is common to `df1` and `df2`: the one with name `Sarah`
- A row is common to `df1` and `df2` yet has a different value for column `Address`: the one with name `Li`
- Some rows exist only in `df1`, or only in `df2`.

Outer merge

Let's use `merge` to gather these datasets in one instance:

In [39]:
```
pd.merge(df1, df2, how='outer', on=['Name', 'Age'], suffixes=('_df1', '_df2'))
```

Out[39]:

	Name	Age	Address_df1	Address_df2
0	Bob	15	Turin	NaN
1	David	45	NaN	Annecy
2	Laura	45	Annecy	NaN
3	Li	23	Chambéry	Paris
4	Pierre	26	NaN	Geneva
5	Sarah	41	Annecy	Annecy

Processing math: 100%

Some explanations:

- `on` tells `pandas` where to look for different tuples of values. These columns must exist in both dataframes.
- `suffixes` makes it possible to assign different names to columns that have the same name in both dataframes.
- `how='outer'` creates one row for every `('Name', 'Age')` pair in `df1` **or** in `df2`.
  - Specifying `how='inner'` would create a row for every pair that exists in `df1` **and** in `df2`
  - `how='left'` only takes pairs of `df1`.
  - `how='right'` only takes pairs of `df2`.

Inner merge

Here after, using `how='inner'`.

In [40]:
```
pd.merge(df1, df2, how='inner', on=['Name', 'Age'], suffixes=('_df1', '_df2'))
```

Out[40]:

	Name	Age	Address_df1	Address_df2
0	Sarah	41	Annecy	Annecy
1	Li	23	Chambéry	Paris

If `on` is set to `'Address'` `how='inner'` only 'Annecy' which is in both `df1` and `df2` is kept:

In [41]:
```
pd.merge(df1, df2, how='inner', on=['Address'], suffixes=('_df1', '_df2'))
```

Out[41]:

	Name_df1	Age_df1	Address	Name_df2	Age_df2
0	Laura	45	Annecy	Sarah	41
1	Sarah	41	Annecy	Sarah	41
2	Laura	45	Annecy	David	45
3	Sarah	41	Annecy	David	45

Left/Right merge

Here after, using `how='left'`.

In [42]:

```
pd.merge(df1, df2, how='left', on=['Name', 'Age'], suffixes=('_df1', '_df2'))
```

Out[42]:

	Name	Age	Address_df1	Address_df2
0	Laura	45	Annecy	NaN
1	Bob	15	Turin	NaN
2	Sarah	41	Annecy	Annecy
3	Li	23	Chambéry	Paris

Applying a rolling function

Suppose we have some experimental data. How can we compute a rolling mean?

In [43]:
```
sr = pd.Series(range(6, 0, -1), index=list('abcdef'))
sr
```

Out[43]:
```
a    6
b    5
c    4
d    3
e    2
f    1
dtype: int64
```

Processing math: 100%

Let's use the `rolling` method:

In [44]:
```
sr.rolling(window=3).mean()
```

Out[44]:
```
a    NaN
b    NaN
c    5.0
d    4.0
e    3.0
f    2.0
dtype: float64
```

The **default behaviour makes the window flushed to the right**: the output value at index k is computed using the input values from $k - windows + 1$ to k.

Processing math: 100%

This baheviour can be changed using `center=True`:

In [45]:
```
sr.rolling(window=3, center=True).mean()
```

Out[45]:
```
a    NaN
b    5.0
c    4.0
d    3.0
e    2.0
f    NaN
dtype: float64
```

Similarly to `groupby` and `resample` objects, one can iterate over what is returned by the `rolling` method:

In [46]:

```
rolling_object = sr.rolling(window=3)
for k in rolling_object:
    print(k)
```

```
a    6
dtype: int64
a    6
b    5
dtype: int64
a    6
b    5
c    4
dtype: int64
b    5
c    4
d    3
dtype: int64
c    4
d    3
e    2
dtype: int64
d    3
e    2
f    1
dtype: int64
```

Processing math: 100%