Data structuring, part III

The Pandas way

Andreas Bjerre-Nielsen

10 things I hate about pandas

- Correction: Integers and NaN do work now!
- Check out this documentation (https://pandas.pydata.org/pandas-docs/stable/user_guide/integer_na.html) from July 2019.

Recap

Which datatypes beyond numeric does pandas handle natively?

- .

What can we do to missing values and duplicates?

-

Agenda

1. the split apply combine framework
2. joining datasets
3. reshaping data
4. methods chaining

Loading the software and data

In [24]:
```
import numpy as np
import pandas as pd
import seaborn as sns

tips = sns.load_dataset('tips')
titanic = sns.load_dataset('titanic')
```

Reshaping data

Stacking data

A DataFrame can be collapsed into a Series with the **stack** command:

In [17]:
```
df = pd.DataFrame([[1,2],[3,4]],columns=['EU','US'],index=[2000,2010])
print(df, '\n')
stacked = df.stack() # going from wide to long format
# stacked=stacked.reset_index()
# stacked.columns = ['year', 'place', 'value']
```

```
      EU  US
2000   1   2
2010   3   4
```

Note: The stacked DataFrame is in long/tidy format, the original is wide.

To wide format

Likewise we can transform a long DataFrame with the unstack

In [45]:
```
print(df.stack())
print()
# print(df.stack().unstack(level=1))
```

```
2000  EU    1
      US    2
2010  EU    3
      US    4
dtype: int64
```

More stuff

Other cool functions include

- `melt` which only stacks certain columns
- `pivot` which makes you to reshape the dataframe like in Excel

Split-apply-combine

Split-apply-combine (1)

What is the split-apply-combine framework?

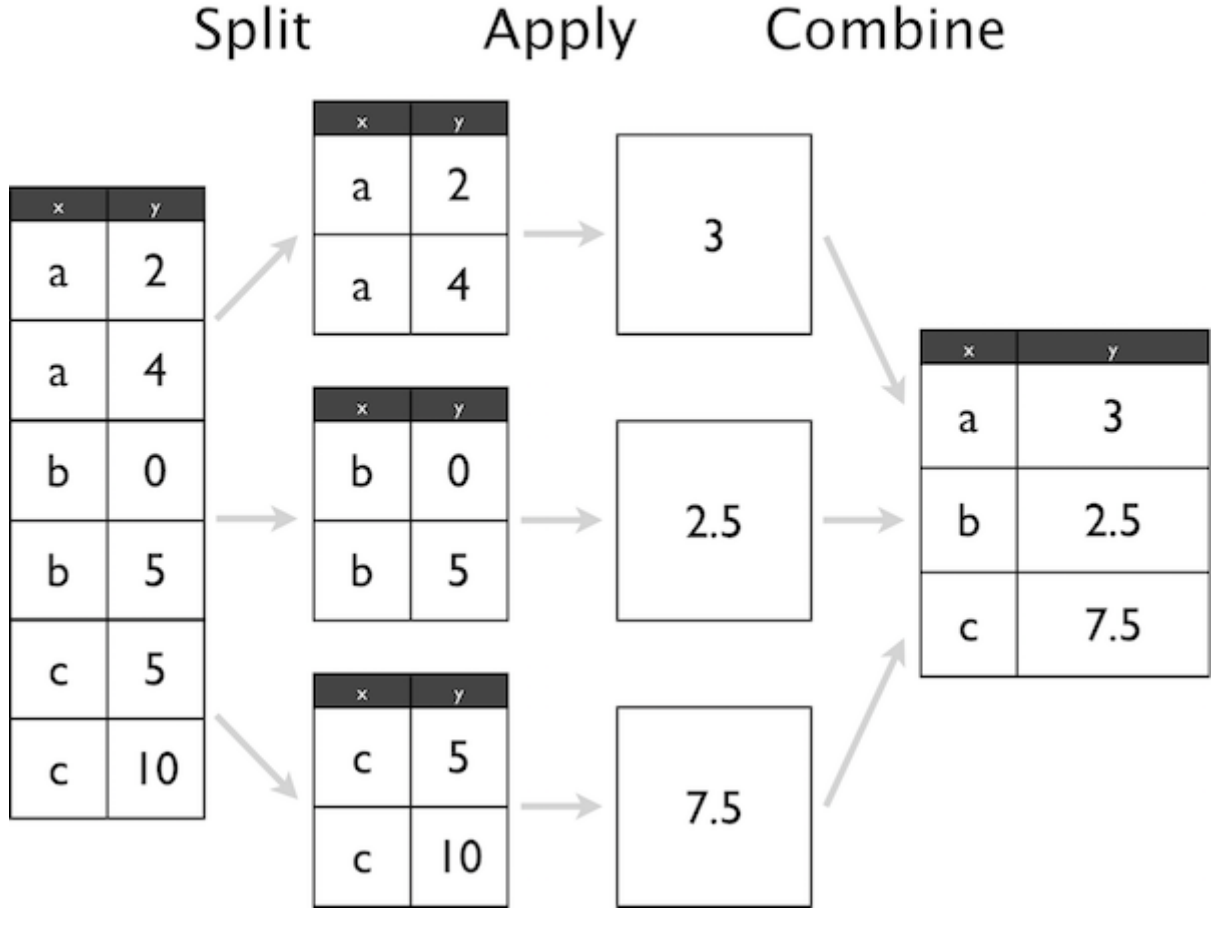
A procedure to

1. **split** a DataFrame into subsets of data
2. **apply** certain functions (sorting, mean, other custom stuff)
3. **combine** it back into a DataFrame

Application example: compute mean personal income.

Split-apply-combine (2)

How do we *split* observations by x and *apply* the calculation mean of y?*



groupby (1)

A powerful tool in DataFrames is the `groupby` method. Example:

In [19]:
```
split_var = 'sex'
apply_var = 'total_bill'

# tips\
#     .groupby(split_var)\
#     [apply_var]\
#     .mean()
```

groupby (2)

What does the `groupby` method do?

- It implements *split-apply-combine*

Can other functions be applied?

- Yes: `mean`, `std`, `min`, `max` all work.
- Using `.apply()` method and inserting your ***homemade*** function works too.

groupby (3)

Does `groupby` work for multiple variables, functions?

In [4]:
```
split_vars = ['sex', 'time']
apply_vars = ['total_bill', 'tip']
apply_fcts = ['mean', 'std', 'median']
combined = tips\
    .groupby(split_vars)\
    [apply_vars]\
    .agg(apply_fcts)

print(combined.reset_index() )
```

```
      sex    time total_bill                      tip
                        mean       std median      mean       std median
0    Male   Lunch  18.048485  7.953435  16.58  2.882121  1.329017   2.31
1    Male  Dinner  21.461452  9.460974  19.63  3.144839  1.529116   3.00
2  Female   Lunch  16.339143  7.500803  13.42  2.582857  1.075108   2.01
3  Female  Dinner  19.213077  8.202085  17.19  3.002115  1.193483   3.00
```

Note grouping with multiple variables uses a [MultiIndex (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.MultiIndex.html)](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.MultiIndex.html) which we do not cover.

groupby (4)

Can we use `groupby` in a loop?

Yes, we can iterate over a groupby object. Example:

In [5]:
```
results = {}
for group, group_df in tips.groupby('sex'):
    group_mean = group_df.total_bill.mean()
    results[group] = group_mean


print(group)
results
```

```
Female
```

Out[5]:
```
{'Female': 18.056896551724137, 'Male': 20.744076433121034}
```

ProTip: `groupby` is an iterable we can also use with `multiprocessing` for parallel computing.

groupby (5)

How do we get our `groupby` output into the original dataframe?

- Option 1: you use `transform`.
- Option 2: you merge it (not recommended)

In [6]:
```
mu_time = tips.groupby(split_var)[apply_var].transform('mean')

(tips.total_bill - mu_time).head()
```

Out[6]:
```
0    -1.066897
1   -10.404076
2     0.265924
3     2.935924
4     6.533103
Name: total_bill, dtype: float64
```

Why is this useful?

- Useful for fixed effects computation

Joining DataFrames

Until now we've worked with one DataFrame at a time.

We will now learn to put them together.

Some DataFrames

Let's make some data to play with

In [38]:
```
left = pd.DataFrame({'key': ['A', 'B', 'C', 'D'], 'value_left': range(4)})
right = pd.DataFrame({'key': ['C', 'D', 'E', 'F'], 'value_right': range(4,8)})
print(left,'\n', right)
```

```
  key  value_left
0   A           0
1   B           1
2   C           2
3   D           3
   key  value_right
0   C            4
1   D            5
2   E            6
3   F            7
```

Merging data

The forthcoming figures all follow this convention:

- blue: rows in merge output
- red: rows excluded from output (i.e., removed)
- green: missing values replaced with NaNs

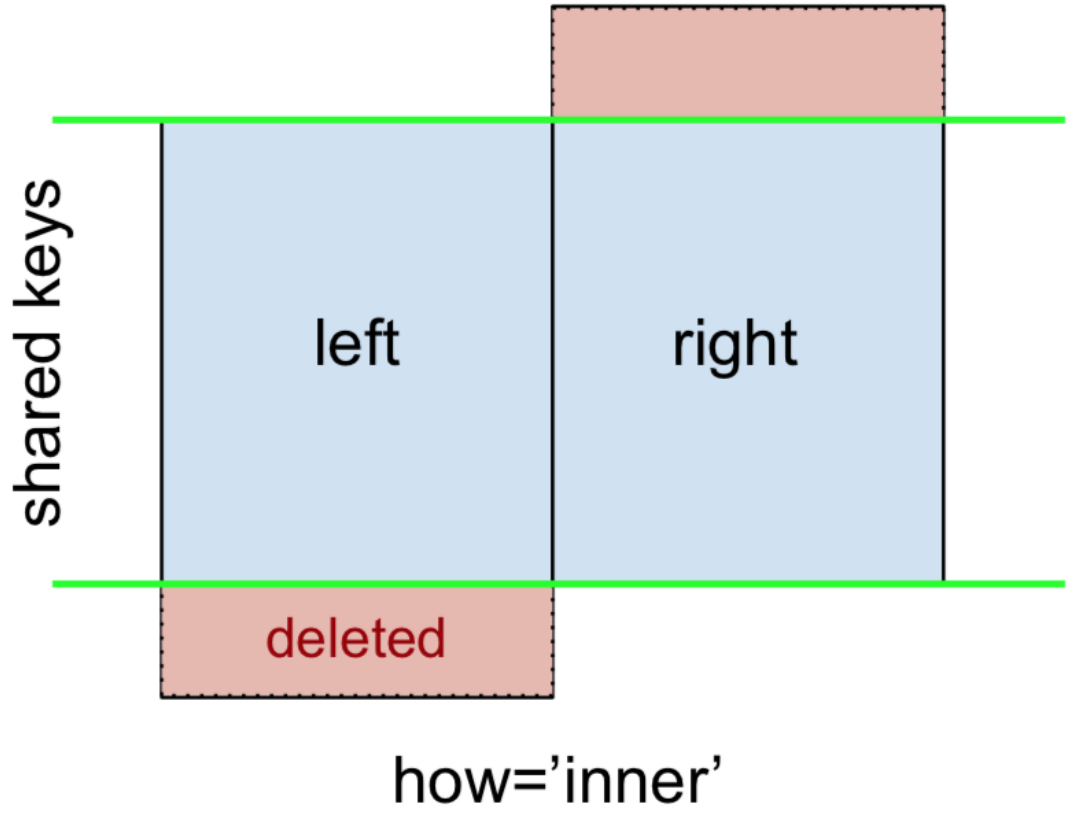
We use `merge` which is pandas function and a method for dataframes.

Inner merge (default)

This merge only uses only *shared* keys

In [39]:
```
print(pd.merge(left, right, on='key', how='inner'))
```

```
  key  value_left  value_right
0   C           2            4
1   D           3            5
```



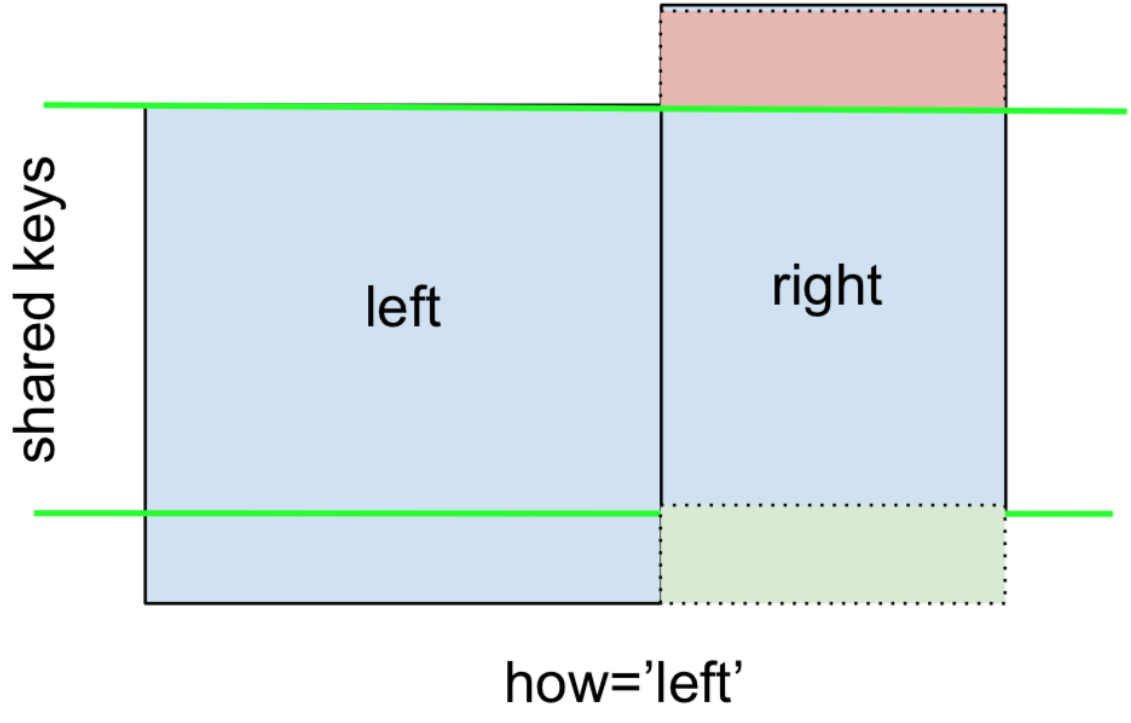
how='inner'

Left merge

This merge uses only *left* keys

In [40]:
```
print(pd.merge(left, right, on='key', how='left'))
```

```
  key  value_left  value_right
0   A           0          NaN
1   B           1          NaN
2   C           2          4.0
3   D           3          5.0
```

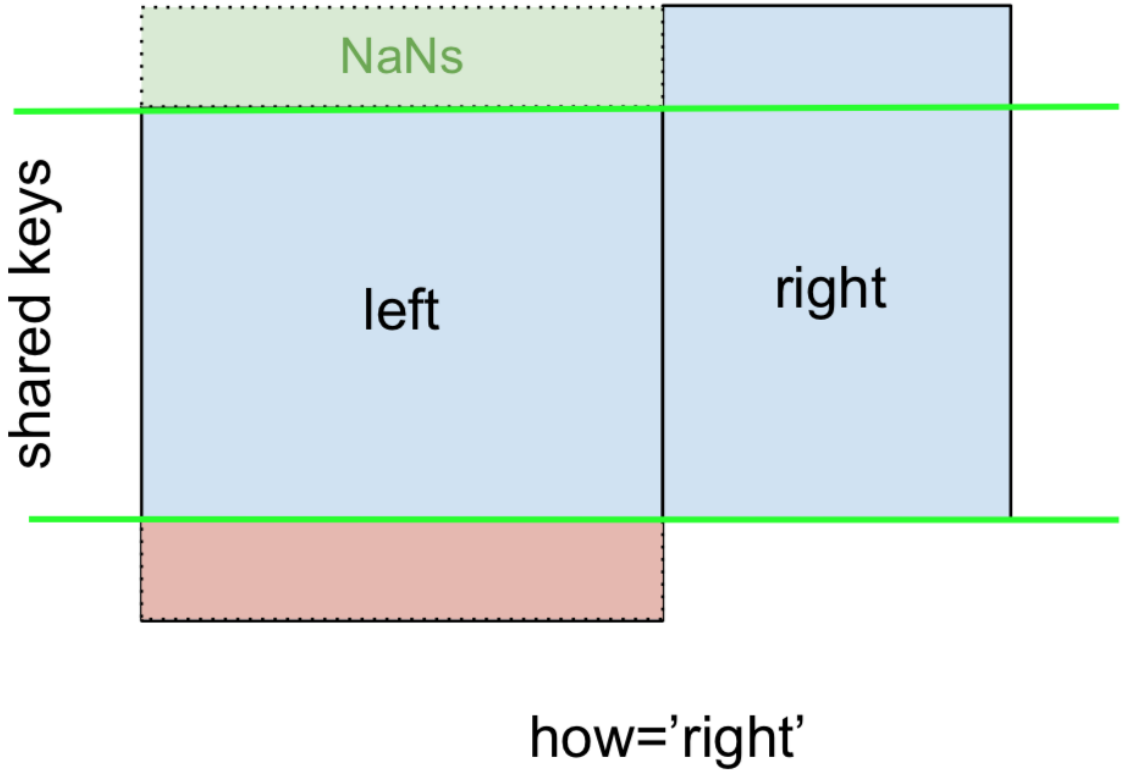

shared keys
left
right
how='left'

Right merge

This merge uses only *right* keys

In [65]:
```
print(pd.merge(left, right, on='key', how='right'))
```

```
  key  value_left  value_right
0   C         2.0            4
1   D         3.0            5
2   E         NaN            6
3   F         NaN            7
```



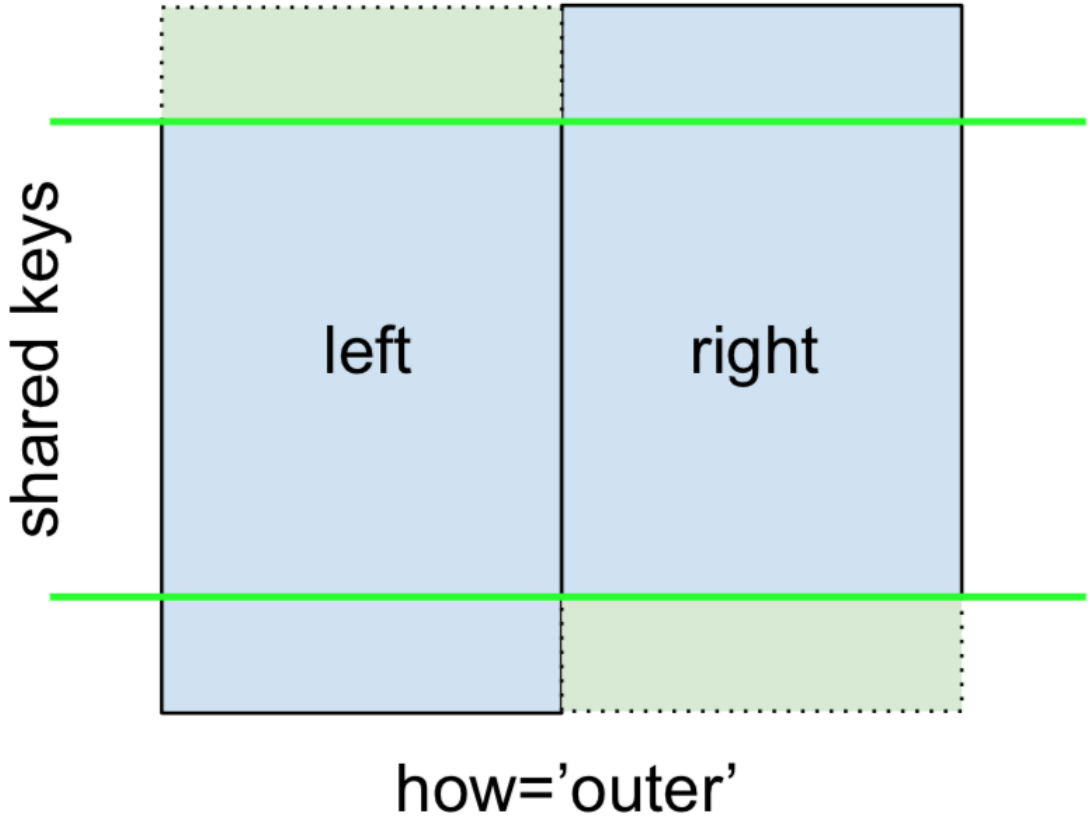
how='right'

Outer merge

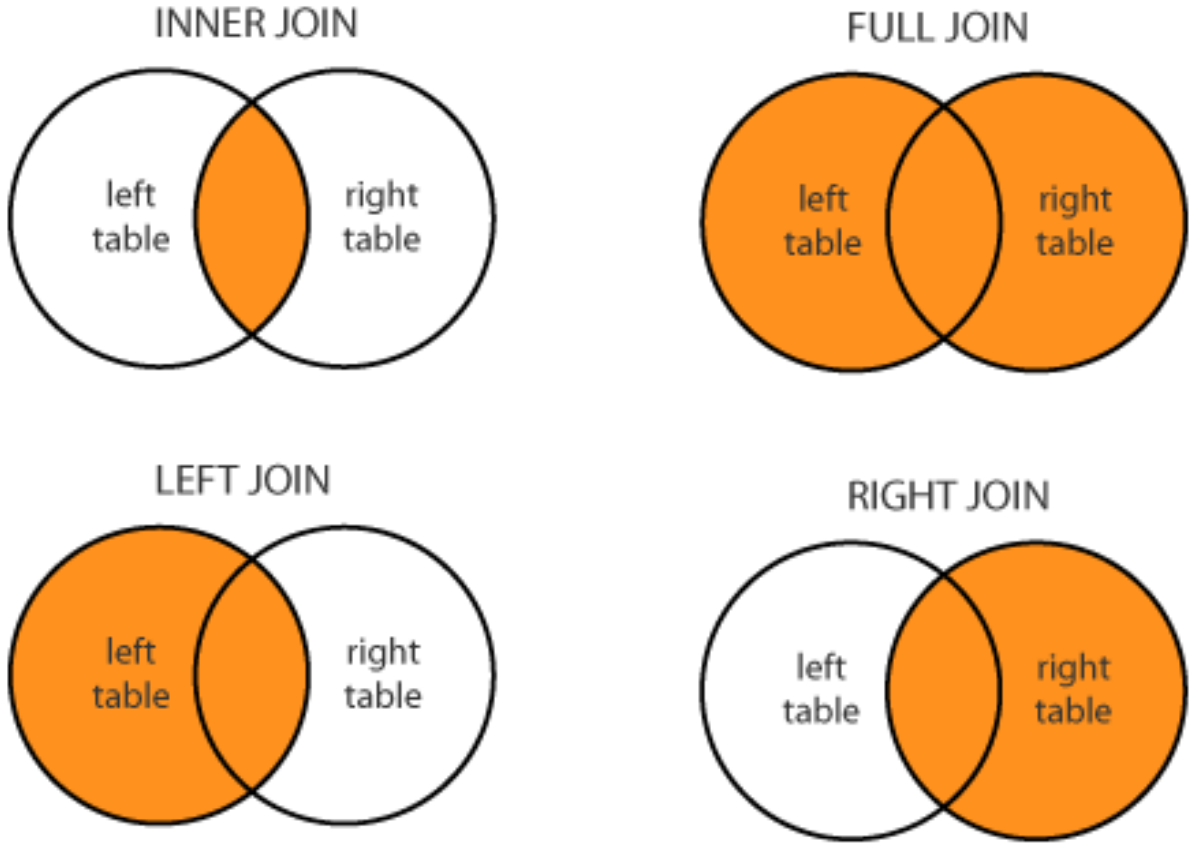
This merge uses *all* keys

In [66]:
```
print(pd.merge(left, right, on='key', how='outer'))
```

```
  key  value_left  value_right
0   A         0.0          NaN
1   B         1.0          NaN
2   C         2.0          4.0
3   D         3.0          5.0
4   E         NaN          6.0
5   F         NaN          7.0
```


shared keys
left
right
how='outer'

Overview of merge types



More merge type exists, see [this post (https://stackoverflow.com/questions/53645882/pandas-merging-101)](https://stackoverflow.com/questions/53645882/pandas-merging-101) for details.

Joining DataFrames

We can also join by keys in the index. This is possible with `join` or `concat`:

- both methods work vertically and horizontally.
- `concat` works with multiple DataFrames at once;

Requirement: overlapping index keys or column names.

In [78]:
```
df0 = left.set_index('key')
df1 = right.set_index('key')
```

Horizontal join

Works like `merge` where keys is now the index!

In [79]:
```
print(df0.join(df1, how='inner'))
```

```
     value_left  value_right
key
C             2            4
D             3            5
```

Vertical join

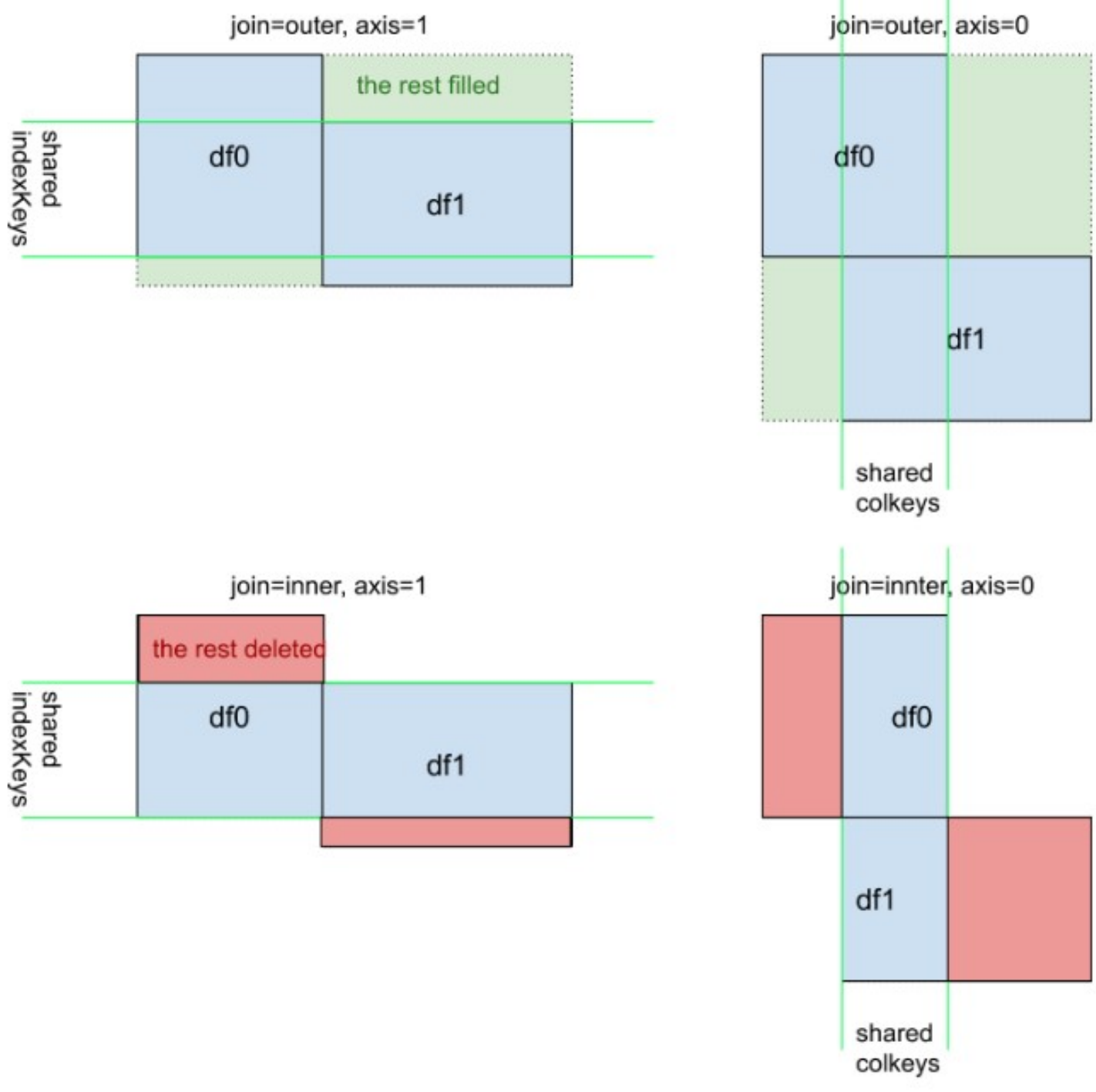
`concat` on axis=0 stacks the dataframes on top of each other!

In [83]:
```
print(pd.concat([df0, df1], join='outer', axis=0, sort=False))
```

```
     value_left  value_right
key
A           0.0          NaN
B           1.0          NaN
C           2.0          NaN
D           3.0          NaN
C           NaN          4.0
D           NaN          5.0
E           NaN          6.0
F           NaN          7.0
```

Vertical and horizontal

An overview of `concat` / `join` operations (left: horizontal, right: vertical)


join=outer, axis=1
the rest filled
shared
indexKeys
df0
df1
join=outer, axis=0
df0
df1
shared
colkeys
join=inner, axis=1
the rest deleted
shared
indexKeys
df0
df1
join=innter, axis=0
df0
df1
shared
colkeys

Putting it together

In [37]:
```
from pandas_datareader import data
stocks = ['aapl', 'goog', 'msft', 'amzn', 'fb']
def load_stock(s):
    return data.DataReader(s, data_source='yahoo', start='2000')['Adj Close']
load_stock('aapl').head()
# stock_dfs = {s:load_stock(s) for s in stocks} # dictionary of all stock price
# stock_df = pd.concat(stock_dfs, axis=1) # horizontal join
# stock_df.plot(logy=True, figsize=(10,3))
```

Out[37]:
```
Date
2000-01-03    3.488905
2000-01-04    3.194754
2000-01-05    3.241507
2000-01-06    2.960991
2000-01-07    3.101249
Name: Adj Close, dtype: float64
```

Methods chaining

Method chain

We iteratively apply methods on dataframes. Example:

In [41]:
```
titanic.groupby('sex').survived.mean()
```

Out[41]:
```
sex
female    0.742038
male      0.188908
Name: survived, dtype: float64
```

Method chain (2)

Suppose we want to filter out teenagers and adults - is this possible?

In [42]:
```
print(titanic.query("age < 13")[['age','sex','survived']].head())
```

```
    age     sex  survived
7   2.0    male         0
10  4.0  female         1
16  2.0    male         0
24  8.0  female         0
43  3.0  female         1
```

Method chain (3)

And how do we make new variables?

In [43]:
```
print(titanic.assign(has_sibsp=lambda df: df.sibsp>0)[['sibsp','has_sibsp']].head(10))
```

```
   sibsp  has_sibsp
0      1       True
1      1       True
2      0      False
3      1       True
4      0      False
5      0      False
6      0      False
7      3       True
8      0      False
9      1       True
```

Method chain (4)

The lines get very long, what do we do?

In [36]:
```
titanic\
    .query("age >= 13")\
    .groupby('sex')\
    .survived\
    .mean()
```

Beyond pandas

If you want more sophisticated data processing tools for big data.

Single machine

- `multiprocessing` and `joblib` for executing code in parallel (using multiple cores)

Multiple machines (cluster)

- `dask` uses a pandas like syntax, also useful for parallelizing
- `pyspark` is Python based but uses a (multiple machines)

The end

Return to agenda