More with pandas

Fabrice Rossi

Examples in this series of exercises are based on the data sets available on the course web page.

Lecture notes

Pandas indexes can have several *levels*. This can be used to implement complex data transformations. The main idea is to have several lists of values of the same size, each of which providing a level of the index. For instance the program

```
import pandas as pd
demo = pd.Series([x**2 for x in range(1,11)],
                 index=[['odd', 'even']*5,
                        [z for z in range(1,11) ]])
print(demo)
```

prints

```
odd   1       1
even  2       4
odd   3       9
even  4      16
odd   5      25
even  6      36
odd   7      49
even  8      64
odd   9      81
even  10    100
dtype: int64
```

The display shows the content of the `Series` (squares of integers) as well as the two levels of the index, a string level and an integer level.

The `Series` can then be accessed using the higher level in a standard way. For instance `demo['odd']` corresponds to

```
1     1
3     9
5    25
7    49
9    81
dtype: int64
```

To access the lower level of the index, one can use successive locators such as `demo[:,2]` which is

```
even    4
```

Exercise 1 (*Hierarchical Index*)

This exercise uses the word data set `'google-10000-english.txt'`[1]. Load it with

```
import pandas as pd
words = pd.read_csv('google-10000-english.txt',
                    names=['word'],
                    header=None)['word']
```

The use of `words.str` to access to the words with a string interface is recommended.

Question 1 Remove words with a single letter from the `Series`.

Question 2 Replace the index of `words` by a hierarchical index with the first letter of the word as the high level index and the second as the low level one.

Question 3 Change the name of the index levels to `'fl'` and `'sl'`, using `words.index.names`.

Question 4 Study the effect of the `sort_index()` method (this is a method of `Series`).

Question 5 Print for each letter the number of words starting with this letter and the number of words with this letter as their second letter. Hint: access to the letters with `words.index.levels[0]`.

Question 6 Study the results of `words.count(level='fl')` and `words.count(level='sl')`. In particular, what is printed by

```
for l, n in words.count(level='fl').items():
    print(l, n)
```

Solution

```
import pandas as pd
words = pd.read_csv('../data/google-10000-english.txt',
                    names=['word'],
                    header=None)['word']
# Q1
words = words[words.str.len() > 1]
# Q2
words.index = [words.str[0], words.str[1]]
# Q3
words.index.names = ['fl', 'sl']
# Q4
words.sort_index()
# Q5
for letter in words.index.levels[0]:
    print(letter,words[letter].size,words[:,letter].size)
# Q6
```

[1] https://github.com/first20hours/google-10000-english

```
print(words.count(level='fl'))
print(words.count(level='sl'))
for l, n in words.count(level='fl').items():
    print(l, n)
```

Lecture notes

As experimented in the first exercise, numerous `Series` methods have a `level` parameter that can be set to the name of a level of the index. The method is applied to all the sub `Series` obtained by selecting one by one all the possible values of the selected level. The result is a `Series` made of the obtained values and indexed by the selected level. For instance

```
import pandas as pd
ms = pd.Series([2*x for x in range(10)],
               index = [[t for t in 'aaabbccccc'],
                        [x%5 for x in range(10)]])
ms.index.names = ['letter', 'digit']
print(ms)
print(ms.sum(level='letter'))
print(ms.sum(level='digit'))
```

prints

```
letter  digit
a       0         0
        1         2
        2         4
b       3         6
        4         8
c       0        10
        1        12
        2        14
        3        16
        4        18
dtype: int64
letter
a     6
b    14
c    70
dtype: int64
digit
0    10
1    14
2    18
3    22
4    26
dtype: int64
```

Exercise 2 (*Hierarchical index and aggregates*)

This exercise uses the french population data set. Load it with

```
import pandas as pd
population = pd.read_csv('population-2014.csv')
```

Question 1 From `population` create a `Series` `nb_com` from the *Nombre de communes*[2] column using columns *Région* and *Département* as the levels of the `Series` index. Hint: use the `values` attribute of the column to extract the values without the index.

Question 2 Using `nb_com`, count the number of *départements* per *Région*.

Question 3 Using `nb_com`, count the total number of cities per *Région*.

Question 4 Obtain the sames results as in the two previous questions using a `groupby` strategy directly on the `population` `DataFrame`.

> **Lecture notes**
>
> A convenient way to build a meaningful hierarchical index in a `DataFrame` is to use some of its variables. For instance the `population` `DataFrame` can be given a global hierarchical index on *Région* and *Département* using
>
> ```
> population = population.set_index(['Région', 'Département'])
> ```
>
> Then the `DataFrame` can be `groupby` using a `level` parameter.

Question 5 Add a variable containing the average population per city for each *département* (using the column *Population municipale*[3]).

Question 6 Compute the average of the average population per city for each *Région*.

Solution

```
import pandas as pd
population = pd.read_csv('../data/population-2014.csv')
nb_com = pd.Series(population['Nombre de communes'].values,
                   index = [population['Région'],
                            population['Département']])
print(nb_com.count(level='Région'))
print(nb_com.sum(level='Région'))
print(population.groupby('Région').size())
print(population.groupby('Région')['Nombre de communes'].sum())
population = population.set_index(['Région', 'Département'])
population['Avg urban pop'] = ( population['Population municipale']
                                / population['Nombre de communes'] )
print(population.groupby(level='Région')['Avg urban pop'].mean())
```

[2] Number of cities.
[3] Urban population.

Exercise 3 (*Joins*)

This exercise uses the financial relational data set[4]. Load the tables into data frames named after the tables, e.g.

```
import pandas as pd
client = pd.read_csv('client.csv')
account = pd.read_csv('account.csv')
```

Question 1 Create a joined table integrating client information into the disposition table.

Question 2 Using the previous table compute the distribution of the account type conditionally on the gender of the user of the account. Hint: see exercise 6 from the "First steps with Pandas" exercise list.

Question 3 Create a `DataFrame` `both_districts` which lists client ids together with the district id of their residence (from the client table) and the district id of their bank (from the account table). Hint: use two joins and take care of identical column names.

Question 4 Using the previous table, select in the client table the clients whose residence is not in the same district as the one of their bank.

Question 5 Compute and print the characteristics of the home district that has the largest number of clients whose bank is not in this district. Hint: use the `idxmax` method of `Series` which gives the index of the maximum value in a `Series`.

Solution

```
import pandas as pd
client = pd.read_csv('../data/Financial/client.csv')
account = pd.read_csv('../data/Financial/account.csv')
district = pd.read_csv('../data/Financial/district.csv')
disposition = pd.read_csv('../data/Financial/disp.csv')
disp_client = pd.merge(disposition, client)
type_gender = disp_client.groupby(['gender','type']).size()
type_gender = type_gender.unstack()
type_gender = type_gender.divide(type_gender.sum(axis=1),axis=0)
both_districts = disp_client[['client_id','account_id','district_id']]
both_districts = both_districts.rename(columns={'district_id': 'home_d_id'})
both_districts = pd.merge(both_districts, account)
client_mismatch = ( client[both_districts['home_d_id']
                      != both_districts['district_id']] )
d_max = client_mismatch.groupby(['district_id']).size().idxmax()
print(district[district['district_id']==d_max])
# with the query method
print(district.query('district_id == @d_max'))
```

[4] https://relational.fit.cvut.cz/dataset/Financial

Lecture notes

Pandas `DataFrame` can be joined using their indexes. While the `merge` function joins by default on common variables, the almost equivalent `join` method joins on common indexes (and thus index names must be set to something meaningful). In the following program, the client table is enriched by district information using index joins.

```
import pandas as pd
client = pd.read_csv('../data/Financial/client.csv')
client.set_index(['district_id','client_id'], inplace=True)
district = pd.read_csv('../data/Financial/district.csv')
district.set_index('district_id', inplace=True)
client_district = client.join(district)
```

Notice that the client table uses a hierarchical index. This is not a requirement. Notice also that the `merge` function can be used to join on indexes (by setting the `left_index` or the `right_index` parameter to `True`).

Exercise 4 (*Joins, indexes and validation*)

This exercise uses the France relational data set. Load it as follows

```
import pandas as pd
city = pd.read_csv('city.csv')
department = pd.read_csv('department.csv')
region = pd.read_csv('region.csv')
```

In France, a city is in a department which is in a region.

Question 1 The columns with names ending in `id` are keys. Identify primary keys and foreign keys. Set the primary keys as index for the corresponding table. Use hierarchical indexes if they make sense.

Question 2 Create a `full_city DataFrame` which contains the cities in `city` complemented with department and region information.

Question 3 Using the `sum` method and its `level` parameter, compute the total population per region and the total population per department.

Solution

```
import pandas as pd
city = pd.read_csv('../data/France/city.csv')
department = pd.read_csv('../data/France/department.csv')
region = pd.read_csv('../data/France/region.csv')
city.set_index(['department_id','city_id'], inplace=True)
department.set_index(['region_id','department_id'],inplace=True)
region.set_index('region_id',inplace=True)
full_city = city.join(department).join(region)
population = full_city['population'].sum(level=['region_id','department_id'])
```

Lecture notes

The `merge` function can be use to validate the structure of the relational data, using the `validation` parameter. The value of the parameter specifies one of the three validation mode:

- `'one_to_one'` (or `'1:1'`) : the join is valid only if the merge keys are unique in both original table;
- `'one_to_many'` (or `'1:m'`) : the join is valid only if the merge keys are unique in the left table;
- `'many_to_one'` (or `'m:1'`) : the join is valid only if the merge keys are unique in the right table.

Question 4 For each interesting join than can be done directly between the three tables of the France data set, chose the stickiest validation mode (it could be none).

Solution

- city with department: many to one
- department with region: many to one

Question 5 Propose a new solution to question 2 that validates the join needed to obtain the `full_city` table.

Solution

```
city_dep = pd.merge(city.reset_index(), department.reset_index(),
                    on='department_id', validate='m:1')
full_city = pd.merge(city_dep, region.reset_index(),
                     on='region_id', validate='m:1')
```