Pandas Tutorial 2: Aggregation and Grouping

Let's continue with the pandas tutorial series. This is the second episode, where I'll introduce aggregation (such as min, max, sum, count, etc.) and grouping. Both are very commonly used methods in analytics and data science projects – so make sure you go through every detail in this!

Note 1: this is a hands-on tutorial, so I recommend doing the coding part with me!

Before we start

If you haven't done so yet, I recommend going through these articles first:

1. How to install Python, R, SQL and bash to practice data science
2. Python for Data Science – Basics #1 – Variables and basic operations
3. Python Import Statement and the Most Important Built-in Modules
4. Top 5 Python Libraries and Packages for Data Scientists

5. Pandas Tutorial 1: Pandas Basics (Reading Data Files, DataFrames, Data Selection

Data aggregation – in theory

Aggregation is the process of turning the values of a dataset (or a subset of it) into one single value. Let me make this clear! If you have a DataFrame like…

animal	water_need
zebra	100
lion	350
elephant	670

kangaroo	200

…then a simple aggregation method is to calculate the summary of the water_needs, which is 100 + 350 + 670 + 200 = 1320. Or a different aggregation method would be to count the number of the animals, which is 4. So the theory is not too complicated. Let's see the rest in practice…

Data aggregation – in practice

Where did we leave off last time? We opened a Jupyter notebook, imported pandas and numpy and loaded two datasets: `zoo.csv` and `article_reads`. We will continue from here – so if you haven't done the "pandas tutorial – episode 1", it's time to go through it!

Okay!

Let's start with our `zoo` dataset! We have loaded it by using:

```
pd.read_csv('zoo.csv', delimiter = ',')
```

```
In [1]: import numpy as np
        import pandas as pd

In [4]: pd.read_csv('zoo.csv', delimiter = ',')
```

Out[4]:

	animal	uniq_id	water_need
0	elephant	1001	500
1	elephant	1002	600
2	elephant	1003	550
3	tiger	1004	300
4	tiger	1005	320
5	tiger	1006	330
6	tiger	1007	290
7	tiger	1008	310
8	zebra	1009	200
9	zebra	1010	220
10	zebra	1011	240

Let's store this dataframe into a variable called `zoo`.

```
zoo = pd.read_csv('zoo.csv', delimiter = ',')
```

```
In [1]: import numpy as np
        import pandas as pd

In [5]: zoo = pd.read_csv('zoo.csv', delimiter = ',')
```

Okay, let's do five things with this data:

1. Let's count the number of rows (the number of animals) in `zoo`!
2. Let's calculate the total `water_need` of the animals!

3. Let's find out which is the smallest `water_need` value!
4. And then the greatest `water_need` value!
5. And eventually the average `water_need`!

Pandas Data Aggregation #1: .count()

Counting the number of the animals is as easy as applying a count function on the `zoo` dataframe:

```
zoo.count()
```

```
In [14]: zoo.count()

Out[14]: animal        22
         uniq_id       22
         water_need    22
         dtype: int64
```

Oh, hey, what are all these lines? Actually, the `.count()` function counts the number of values in *each column*. In the case of the `zoo` dataset, there were 3 columns, and each of them had 22 values in it.

If you want to make your output clearer, you can select the `animal` column first by using one of the selection operators from the previous article:

```
zoo[['animal']].count()
```

```
In [17]: zoo[['animal']].count()
Out[17]: animal    22
         dtype: int64
```

Or in this particular case, the result could be even nicer if you use this syntax:

```
zoo.animal.count()
```

This also selects only one column, but it turns our pandas dataframe object into a pandas series object. *(Which means that the output format is slightly different.)*

```
In [15]: zoo.animal.count()
Out[15]: 22
```

Pandas Data Aggregation #2: .sum()

Following the same logic, you can easily sum the values in the `water_need` column by typing:

```
zoo.water_need.sum()
```

```
In [19]: zoo.water_need.sum()
Out[19]: 7650
```

Just out of curiosity, let's run our sum function on all columns, as well:

```
zoo.sum()
```

```
In [20]: zoo.sum()
Out[20]: animal        elephantelephantelephanttigertigertigertigerti...
         uniq_id                                                   22253
         water_need                                                 7650
         dtype: object
```

Note: I love how `.sum()` *turns the words of the* `animal` *column into one string of animal names. (By the way, it's very much in line with the logic of Python.)*

Pandas Data Aggregation #3 and #4: .min() and .max()

What's the smallest value in the `water_need` column? I bet you have figured it out already:

```
zoo.water_need.min()
```

```
In [21]: zoo.water_need.min()
Out[21]: 80
```

And the max value is pretty similar:

```
zoo.water_need.max()
```

```
In [22]: zoo.water_need.max()
Out[22]: 600
```

Pandas Data aggregation #5 and #6: .mean() and .median()

Eventually, let's calculate statistical averages, like mean and median:

```
zoo.water_need.mean()
```

```
In [25]: zoo.water_need.mean()
Out[25]: 347.72727272727275
```

`zoo.water_need.median()`

```
In [26]: zoo.water_need.median()
Out[26]: 325.0
```

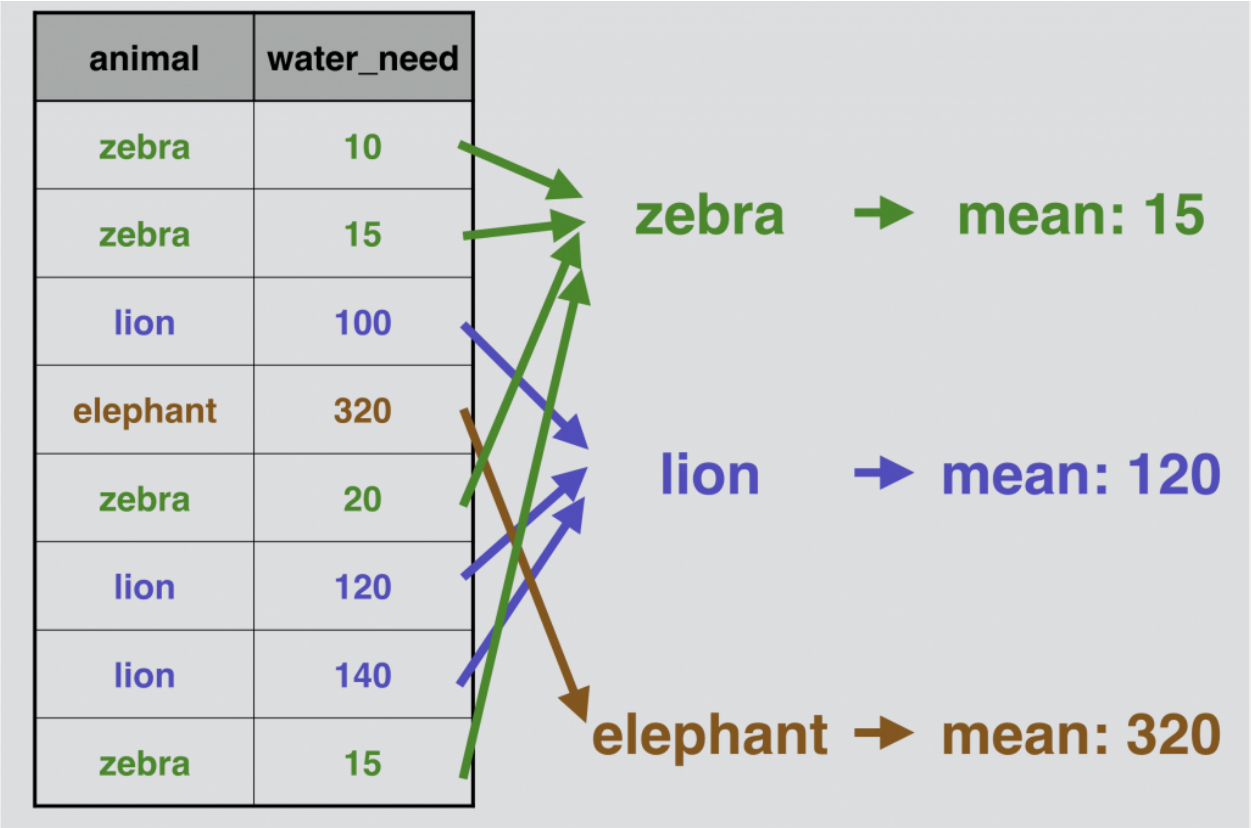
Okay, this was easy. Much, much easier than the aggregation methods of SQL.

But let's spice this up with a little bit of grouping.

Grouping in pandas

As a Data Analyst or Scientist you will probably do segmentations all the time. For instance, it's nice to know the mean `water_need` of all animals (we have just learned that it's `347.72`). But very often it's much more actionable to break this number down – let's say – by animal types. With that, we can compare the species to each other – or we can find outliers.

Here's a simplified visual that shows how pandas performs "segmentation" (grouping and aggregation) based on the column values!



Pandas .groupby in action

Let's do the above presented grouping and aggregation for real, on our `zoo` DataFrame!

We have to fit in a groupby keyword between our `zoo` variable and our `.mean()` function:

```
zoo.groupby('animal').mean()
```

```
In [32]: zoo.groupby('animal').mean()
```

Out[32]:

animal	uniq_id	water_need
elephant	1002.0	550.000000
kangaroo	1021.0	416.666667
lion	1017.5	477.500000
tiger	1006.0	310.000000
zebra	1012.0	184.285714

Just as before, pandas automatically runs the `.mean()` calculation for all remaining columns (the `animal` column obviously disappeared, since that was the column we grouped by). You can either ignore the `uniq_id` column, or you can remove it afterwards by using one of these syntaxes:

`zoo.groupby('animal').mean()[['water_need']]` –» This returns a DataFrame object.

`zoo.groupby('animal').mean().water_need` –» This returns a Series object.

```
In [39]: zoo.groupby('animal').mean()[['water_need']]
```

Out[39]:

animal	water_need
elephant	550.000000
kangaroo	416.666667
lion	477.500000
tiger	310.000000
zebra	184.285714

Obviously, you can change the aggregation method from `.mean()` to anything we learned above!

Okay! Now you know everything, you have to know!

It's time to…

Test yourself #1

Let's get back to our `article_read` dataset.

```
In [46]: article_read
```

Out[46]:

	my_datetime	event	country	user_id	source	topic
0	2018-01-01 00:01:01	read	country_7	2458151261	SEO	North America
1	2018-01-01 00:03:20	read	country_7	2458151262	SEO	South America
2	2018-01-01 00:04:01	read	country_7	2458151263	AdWords	Africa
3	2018-01-01 00:04:02	read	country_7	2458151264	AdWords	Europe
4	2018-01-01 00:05:03	read	country_8	2458151265	Reddit	North America
5	2018-01-01 00:05:42	read	country_6	2458151266	Reddit	North America
6	2018-01-01 00:06:06	read	country_2	2458151267	Reddit	Europe
7	2018-01-01 00:06:15	read	country_6	2458151268	AdWords	Europe
8	2018-01-01 00:07:21	read	country_7	2458151269	AdWords	North America
9	2018-01-01 00:07:29	read	country_5	2458151270	Reddit	North America
10	2018-01-01 00:07:57	read	country_5	2458151271	AdWords	Asia

If you have everything set, here's my first assignment:

What's the most frequent source in the `article_read` dataframe?

.

.

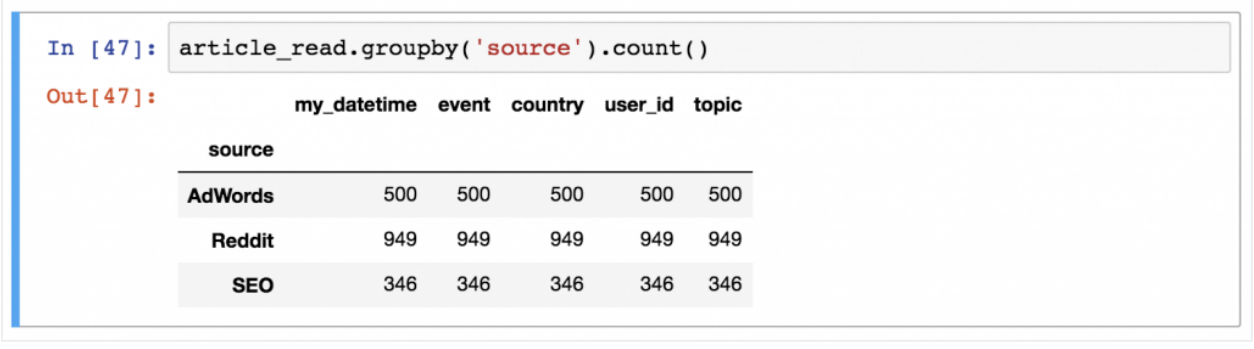
.

And the solution is: ***Reddit***!

How did I get it? Use this code:

```
article_read.groupby('source').count()
```

Take the `article_read` dataset, create segments by the values of the `source` column (`groupby('source')`), and eventually count the values by sources (`.count()`).



```
In [47]: article_read.groupby('source').count()
```

Out[47]:

source	my_datetime	event	country	user_id	topic
AdWords	500	500	500	500	500
Reddit	949	949	949	949	949
SEO	346	346	346	346	346

You can – optionally – remove the unnecessary columns and keep the `user_id` column only:

```
article_read.groupby('source').count()[['user_id']]
```

Test yourself #2

Here's another, slightly more complex challenge:

For the users of `country_2`, what was the most frequent topic and source combination? Or in other words: which topic, from which source, brought the most views from country_2?

.

.

.

The result is: the combination of Reddit (source) and Asia (topic), with 139 reads!

And the Python code to get this results is:

```
article_read[article_read.country == 'country_2'].groupby(['source', 'topic']).count()
```

```
In [58]: article_read[article_read.country == 'country_2'].groupby(['source', 'topic']).count()
```

Out[58]:

source	topic	my_datetime	event	country	user_id
AdWords	Africa	3	3	3	3
	Asia	31	31	31	31
	Australia	6	6	6	6
	Europe	46	46	46	46
	North America	11	11	11	11
	South America	14	14	14	14
Reddit	Africa	24	24	24	24
	Asia	139	139	139	139
	Australia	18	18	18	18
	Europe	29	29	29	29
	North America	27	27	27	27
	South America	26	26	26	26
SEO	Africa	7	7	7	7
	Asia	9	9	9	9
	Australia	10	10	10	10
	Europe	4	4	4	4
	North America	42	42	42	42
	South America	16	16	16	16

Here's a brief explanation:

First, we filtered for the users of `country_2` (`article_read[article_read.country == 'country_2']`). Then on this subset, we applied a `groupby` pandas method… Oh, did I mention that you can group by multiple columns? Now you know that! (Syntax-wise, watch out for one thing: you have to put the name of the columns into a list. That's why the bracket frames go between the parentheses.) (That was the `groupby(['source', 'topic'])` part.)

And as per usual: the `count()` function is the last piece of the puzzle.

Conclusion

This was the second episode of my pandas tutorial series. I hope now you see that aggregation and grouping is really easy and straightforward in pandas… and believe me, you will use them a lot!

Note: If you have used SQL before, I encourage you to take a break and compare the pandas and the SQL methods of aggregation. With that you will understand more about the key differences between the two languages!

In the next, I'll show you the four most commonly used "data wrangling" methods: `merge`, `sort`, `reset_index` and `fillna`