Pandas Tutorial 3: Important Data Formatting Methods (merge, sort, reset_index, fillna)

Written by Tomi Mester on August 13, 2018

This is the third episode of my pandas tutorial series. In this one I'll show you four data formatting methods that you might use a lot in data science projects. These are: `merge`, `sort`, `reset_index` and `fillna`! Of course, there are many others, and at the end of the article, I'll link to a pandas cheat sheet where you can find every function and method you could ever need. Okay! Let's get started!

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

Note 2: some of my code examples will be broken into more lines (by my blog engine) to fit your screen, while in your Jupyter Notebook they should be written in one line. If you copy-paste, it won't be a problem – but if you don't: watch out for these lines! By the way, if you made it this far with my tutorials, I'm pretty sure, you can handle this issue easily by yourself (but screenshots are also there to help you.) 😉

Before we start

1. How to install Python, R, SQL and bash to practice data science
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)
6. Pandas Tutorial 2: Aggregation and Grouping

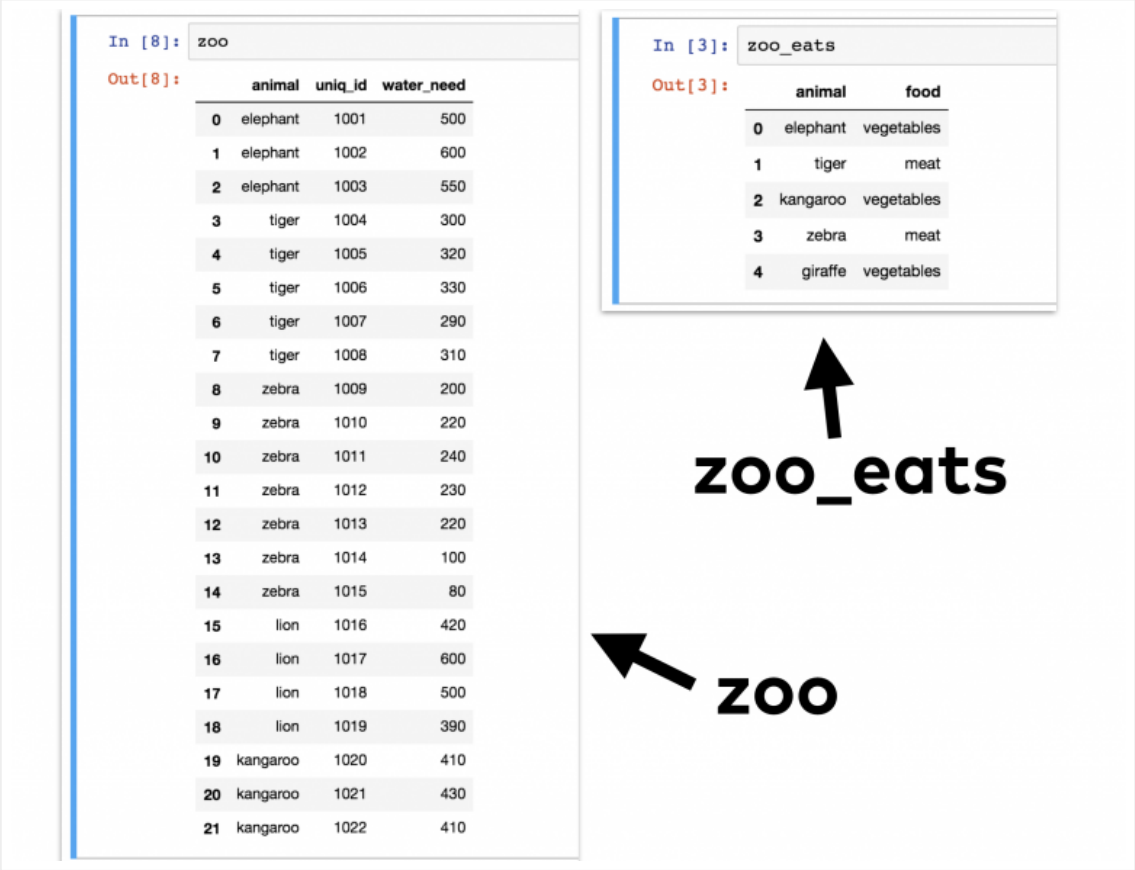
Pandas Merge (a.k.a. "joining" dataframes)

In real life data projects, we usually don't store all the data in one big data table. We store it in a few smaller ones instead. There are many reasons behind this; by using multiple data tables, it's easier to manage your data, it's easier to avoid redundancy, you can save some disk space, you can query the smaller tables faster, etc.

The point is that it's quite usual that during your analysis you have to pull your data from two or more different tables. The solution for that is called **merge**.

Note: Although it's called merge in pandas, it's almost the same as SQL's JOIN method.

Let me show you an example! Let's take our `zoo` dataframe (from our previous tutorials) in which we have all our animals… and let's say that we have another dataframe, `zoo_eats`, that contains information about the food requirements for each species.



We want to merge these two pandas dataframes into one big dataframe. Something like this:

Out[4]:

	animal	uniq_id	water_need	food
0	elephant	1001	500	vegetables
1	elephant	1002	600	vegetables
2	elephant	1003	550	vegetables
3	tiger	1004	300	meat
4	tiger	1005	320	meat
5	tiger	1006	330	meat
6	tiger	1007	290	meat
7	tiger	1008	310	meat
8	zebra	1009	200	meat
9	zebra	1010	220	meat
10	zebra	1011	240	meat
11	zebra	1012	230	meat
12	zebra	1013	220	meat
13	zebra	1014	100	meat
14	zebra	1015	80	meat
15	kangaroo	1020	410	vegetables
16	kangaroo	1021	430	vegetables
17	kangaroo	1022	410	vegetables

In this table, it's finally possible to analyze, for instance, how many animals in our zoo eat meat or vegetables.

How did I do the merge?

First of all, you have the `zoo` dataframe already, but for this exercise you will have to create a `zoo_eats` dataframe, too. For your convenience, here's the raw data of the `zoo_eats` dataframe:

```
animal;food
elephant;vegetables
tiger;meat
kangaroo;vegetables
zebra;vegetables
giraffe;vegetables
```

If I were you, to put this into a proper pandas dataframe, I'd follow the process from the Pandas Tutorial 1 article, but if you want to do this the lazy way, here's a shortcut. 😉 Just copy-paste this (really long) one line into the *pandas_tutorial_1* Jupyter Notebook we made in the first Pandas tutorial:

```
zoo_eats = pd.DataFrame([['elephant','vegetables'],
['tiger','meat'], ['kangaroo','vegetables'],
['zebra','vegetables'], ['giraffe','vegetables']], columns=
['animal', 'food'])
```

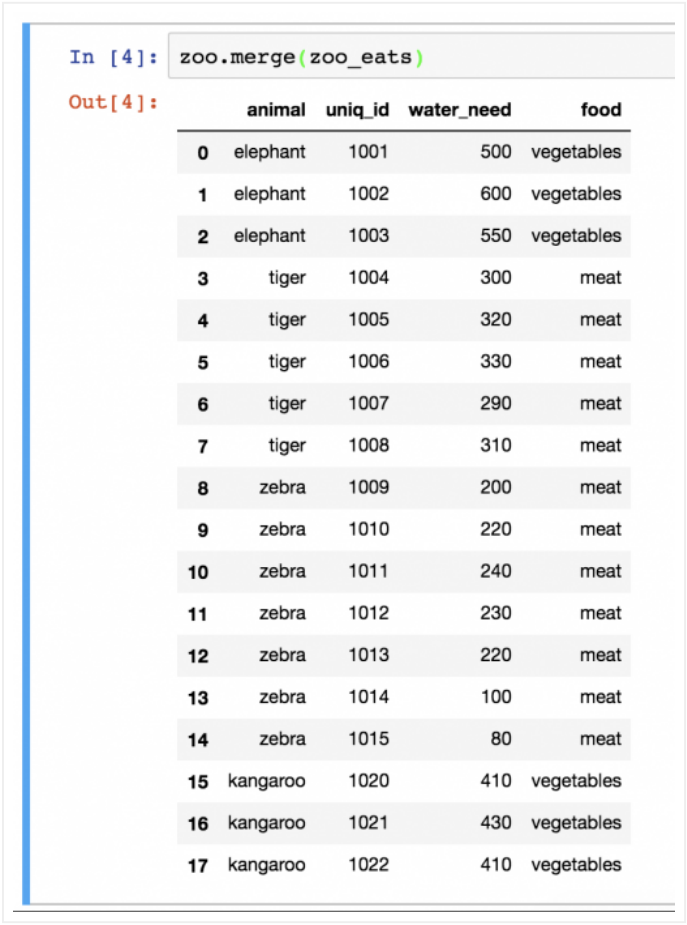
```
In [5]: zoo_eats = pd.DataFrame([['elephant','vegetables'], ['tiger','meat'], ['kangaroo','vegetables'], ['zebra','vegetables']
In [6]: zoo_eats
Out[6]:
```

	animal	food
0	elephant	vegetables
1	tiger	meat
2	kangaroo	vegetables
3	zebra	vegetables
4	giraffe	vegetables

And there is your `zoo_eats` dataframe!

Okay, now let's see the pandas merge method:

```
zoo.merge(zoo_eats)
```



```
In [4]: zoo.merge(zoo_eats)
Out[4]:
```

	animal	uniq_id	water_need	food
0	elephant	1001	500	vegetables
1	elephant	1002	600	vegetables
2	elephant	1003	550	vegetables
3	tiger	1004	300	meat
4	tiger	1005	320	meat
5	tiger	1006	330	meat
6	tiger	1007	290	meat
7	tiger	1008	310	meat
8	zebra	1009	200	meat
9	zebra	1010	220	meat
10	zebra	1011	240	meat
11	zebra	1012	230	meat
12	zebra	1013	220	meat
13	zebra	1014	100	meat
14	zebra	1015	80	meat
15	kangaroo	1020	410	vegetables
16	kangaroo	1021	430	vegetables
17	kangaroo	1022	410	vegetables

(Oh, hey, where are all the lions? We will get back to that soon, I promise!)

Bamm! Simple, right? Just in case, let's see what's happening here:

First, I specified the first dataframe (`zoo`), then I applied the `.merge()` pandas method on it and as a parameter I specified the second dataframe (`zoo_eats`). I could have done this the other way around:

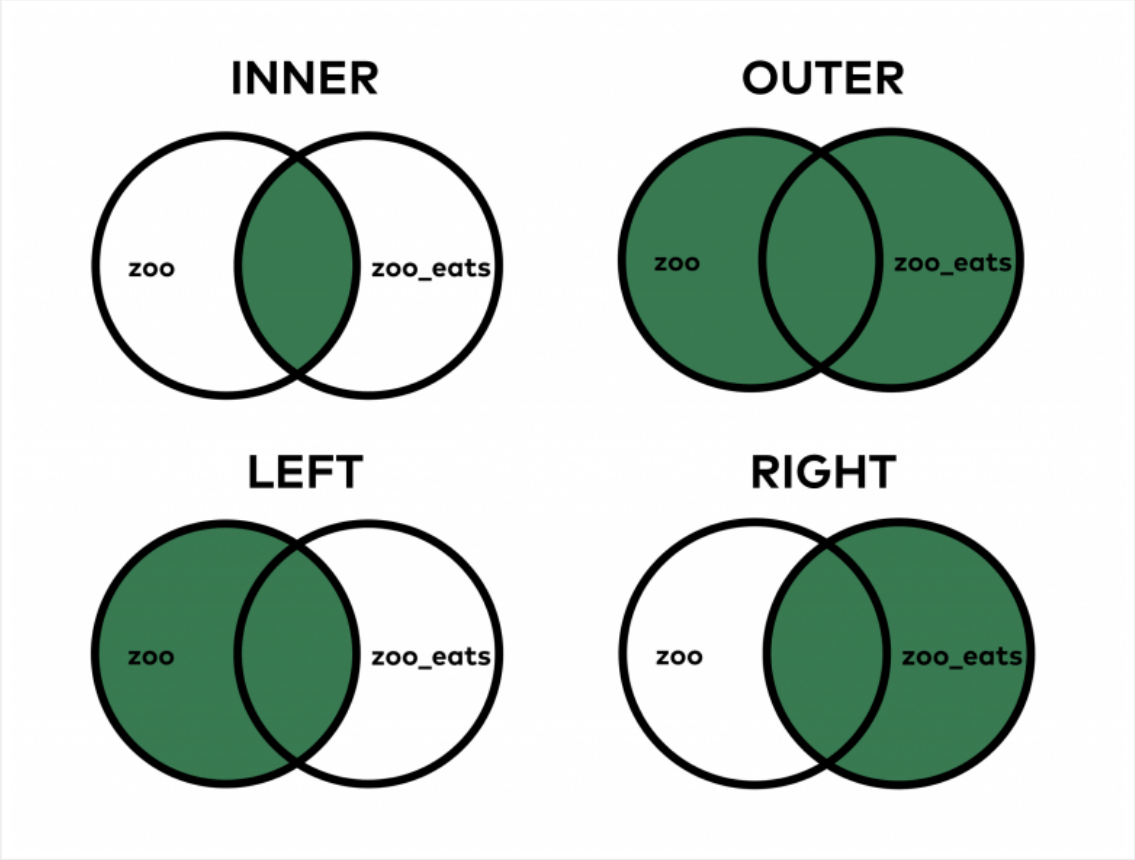
`zoo_eats.merge(zoo)`
is symmetric to:
`zoo.merge(zoo_eats)`

The only difference between the two is the order of the columns in the output table. (Just try it!)

Pandas Merge… But how? Inner, outer, left or right?

As you can see, the basic merge method is pretty simple. Sometimes you have to add a few extra parameters though.

One of the most important questions is ***how*** you want to merge these tables. In SQL, we learned that there are different JOIN types.



SQL JOIN types and pandas merge types

The theory is exactly the same for pandas merge.

When you do an INNER JOIN (that's the default both in SQL and pandas), you merge only those values that are found in **both tables**. On the other hand, when you do the OUTER JOIN, it merges all values, even if you can find some of them in only one of the tables.

Let's see a concrete example: did you realize that there is no `lion` value in `zoo_eats`? Or that we don't have any giraffes in `zoo`? When we did the `merge` above, by default, it was an INNER merge, so it filtered out giraffes and lions from the result table. But there are cases in which we do want to see these values in our joined dataframe. Let's try this:

```
zoo.merge(zoo_eats, how = 'outer')
```

```
In [11]: zoo.merge(zoo_eats, how = 'outer')
```

Out[11]:

	animal	uniq_id	water_need	food
0	elephant	1001.0	500.0	vegetables
1	elephant	1002.0	600.0	vegetables
2	elephant	1003.0	550.0	vegetables
3	tiger	1004.0	300.0	meat
4	tiger	1005.0	320.0	meat
5	tiger	1006.0	330.0	meat
6	tiger	1007.0	290.0	meat
7	tiger	1008.0	310.0	meat
8	zebra	1009.0	200.0	vegetables
9	zebra	1010.0	220.0	vegetables
10	zebra	1011.0	240.0	vegetables
11	zebra	1012.0	230.0	vegetables
12	zebra	1013.0	220.0	vegetables
13	zebra	1014.0	100.0	vegetables
14	zebra	1015.0	80.0	vegetables
15	lion	1016.0	420.0	NaN
16	lion	1017.0	600.0	NaN
17	lion	1018.0	500.0	NaN
18	lion	1019.0	390.0	NaN
19	kangaroo	1020.0	410.0	vegetables
20	kangaroo	1021.0	430.0	vegetables
21	kangaroo	1022.0	410.0	vegetables
22	giraffe	NaN	NaN	vegetables

See? Lions came back, the giraffe came back… The only thing is that we have empty (`NaN`) values in those columns where we didn't get information from the other table.

In my opinion, in this specific case, it would make more sense to keep lions in the table but not the giraffes… With that, we could see all the animals in our zoo and we would have three food categories: `vegetables` , `meat` and `NaN` (which is basically "no information"). Keeping the giraffe line would be misleading and irrelevant since we don't have any giraffes in our zoo anyway. That's when merging with a `how = 'left'` parameter becomes handy!

Try this:

```
zoo.merge(zoo_eats, how = 'left')
```

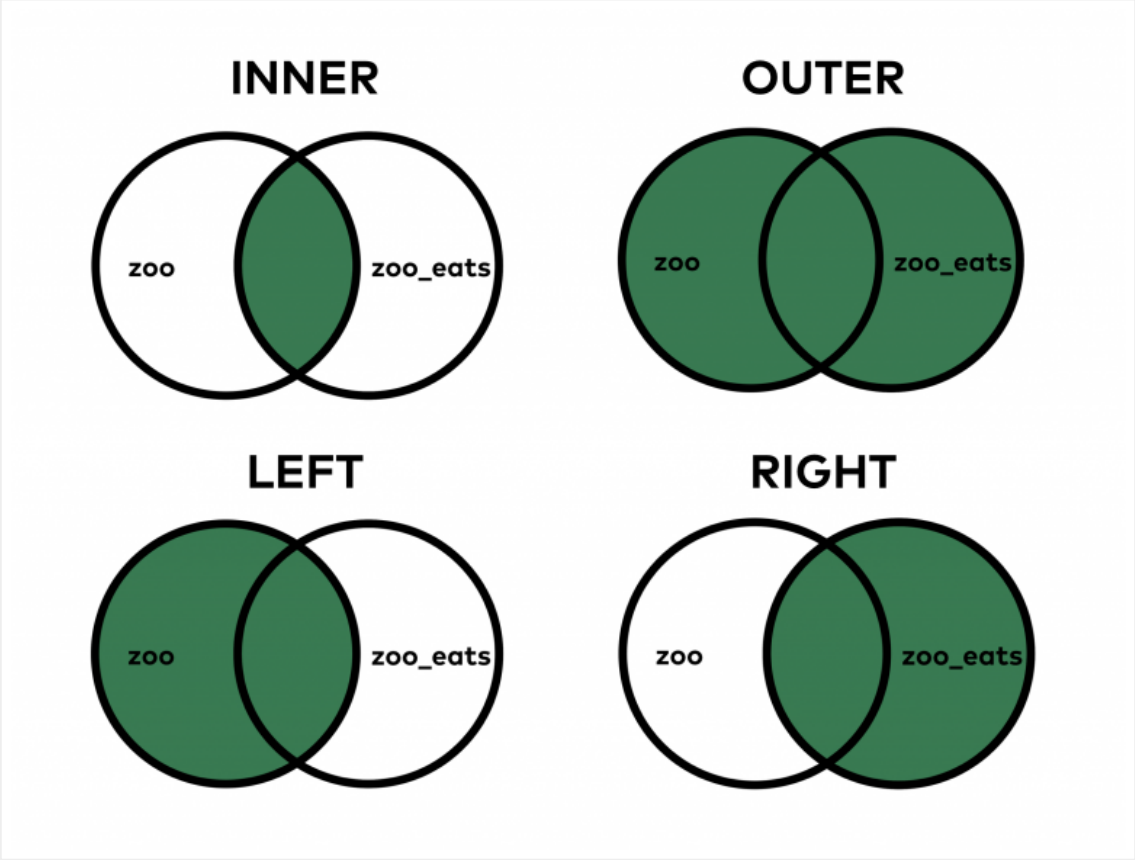
```
In [12]: zoo.merge(zoo_eats, how = 'left')
```

Out[12]:

	animal	uniq_id	water_need	food
0	elephant	1001	500	vegetables
1	elephant	1002	600	vegetables
2	elephant	1003	550	vegetables
3	tiger	1004	300	meat
4	tiger	1005	320	meat
5	tiger	1006	330	meat
6	tiger	1007	290	meat
7	tiger	1008	310	meat
8	zebra	1009	200	vegetables
9	zebra	1010	220	vegetables
10	zebra	1011	240	vegetables
11	zebra	1012	230	vegetables
12	zebra	1013	220	vegetables
13	zebra	1014	100	vegetables
14	zebra	1015	80	vegetables
15	lion	1016	420	NaN
16	lion	1017	600	NaN
17	lion	1018	500	NaN
18	lion	1019	390	NaN
19	kangaroo	1020	410	vegetables
20	kangaroo	1021	430	vegetables
21	kangaroo	1022	410	vegetables

Everything you do need, and nothing you don't… The `how = 'left'` parameter brought all the values from the left table (`zoo`) but brought only those values from the right table (`zoo_eats`) that we have in the left one, too. Cool!

Let's take a look at our merge types again:



Note: a common question I get is "What's the "safest" way of merging? Should you go with inner, outer, left or right, as a best practice?" My answer is: there is no categorical answer for this question. While inner is the default merge type in pandas, whether you should go with that, or change to outer, left or right, really depends on the task itself.

Pandas Merge. On which column?

For doing the merge, pandas needs the key-columns you want to base the merge on (in our case it was the `animal` column in both tables). If you are not so lucky that pandas automatically recognizes these key-columns, you have to help it by providing the column names. That's what the `left_on` and `right_on` parameters are for!

For example, our latest left merge could have looked like this, as well:

```
zoo.merge(zoo_eats, how = 'left', left_on = 'animal', right_on = 'animal')
```

Note: again, in the previous examples pandas automatically found the key-columns anyway… but there are many cases when it doesn't. So keep `left_on` *and* `right_on` *in mind.*

Okay, pandas merge was quite complex; the rest of the methods I'll show you here will be much easier.

Sorting in pandas

Sorting is essential. The basic sorting method is not too difficult in pandas. The function is called `sort_values()` and it works like this:

```
zoo.sort_values('water_need')
```

```
In [18]: zoo.sort_values('water_need')
```

Out[18]:

	animal	uniq_id	water_need
14	zebra	1015	80
13	zebra	1014	100
8	zebra	1009	200
9	zebra	1010	220
12	zebra	1013	220
11	zebra	1012	230
10	zebra	1011	240
6	tiger	1007	290
3	tiger	1004	300
7	tiger	1008	310
4	tiger	1005	320
5	tiger	1006	330
18	lion	1019	390
19	kangaroo	1020	410
21	kangaroo	1022	410
15	lion	1016	420
20	kangaroo	1021	430
17	lion	1018	500
0	elephant	1001	500
2	elephant	1003	550
16	lion	1017	600
1	elephant	1002	600

Note: in the older version of pandas, there is a `sort()` *function with a similar mechanism. But it has been replaced with* `sort_values()` *in newer versions, so learn* `sort_values()` *and not* `sort()`.

The only parameter I used here was the name of the column I want to sort by, in this case the `water_need` column. Quite often, you have to sort by multiple columns, so in general, I recommend using the `by` keyword for the columns:

```
zoo.sort_values(by = ['animal', 'water_need'])
```

```
In [22]: zoo.sort_values(by = ['animal', 'water_need'])
Out[22]:
```

	animal	uniq_id	water_need
0	elephant	1001	500
2	elephant	1003	550
1	elephant	1002	600
19	kangaroo	1020	410
21	kangaroo	1022	410
20	kangaroo	1021	430
18	lion	1019	390
15	lion	1016	420
17	lion	1018	500
16	lion	1017	600
6	tiger	1007	290
3	tiger	1004	300
7	tiger	1008	310
4	tiger	1005	320
5	tiger	1006	330
14	zebra	1015	80
13	zebra	1014	100
8	zebra	1009	200
9	zebra	1010	220
12	zebra	1013	220
11	zebra	1012	230
10	zebra	1011	240

Note: you can use the by keyword with one column only, too, like `zoo.sort_values(by = ['water_need'])`.

`sort_values` sorts in ascending order, but obviously, you can change this and do descending order as well:

```
zoo.sort_values(by = ['water_need'], ascending = False)
```

```
In [25]: zoo.sort_values(by = ['water_need'], ascending = False)
```

Out[25]:

	animal	uniq_id	water_need
1	elephant	1002	600
16	lion	1017	600
2	elephant	1003	550
0	elephant	1001	500
17	lion	1018	500
20	kangaroo	1021	430
15	lion	1016	420
19	kangaroo	1020	410
21	kangaroo	1022	410
18	lion	1019	390
5	tiger	1006	330
4	tiger	1005	320
7	tiger	1008	310
3	tiger	1004	300
6	tiger	1007	290
10	zebra	1011	240
11	zebra	1012	230
9	zebra	1010	220
12	zebra	1013	220
8	zebra	1009	200
13	zebra	1014	100
14	zebra	1015	80

Am I the only one who finds it funny that defining descending is possible only as `ascending = False`? Whatever. 🙂

Reset_index

(This section is especially important for you if you participate in the Junior Data Scientist's First Month video course.)

What a mess with all the indexes after that last sorting, right?

```
In [25]: zoo.sort_values(by = ['water_need'], ascending = False)
```

Out[25]:

	animal	uniq_id	water_need
1	elephant	1002	600
16	lion	1017	600
2	elephant	1003	550
0	elephant	1001	500
17	lion	1018	500
20	kangaroo	1021	430
15	lion	1016	420
19	kangaroo	1020	410
21	kangaroo	1022	410
18	lion	1019	390
5	tiger	1006	330
4	tiger	1005	320
7	tiger	1008	310
3	tiger	1004	300
6	tiger	1007	290
10	zebra	1011	240
11	zebra	1012	230
9	zebra	1010	220
12	zebra	1013	220
8	zebra	1009	200
13	zebra	1014	100
14	zebra	1015	80

It's not just that it's ugly… wrong indexing can mess up your visualizations (more about that in my matplotlib tutorials) or even your machine learning models.

The point is: in certain cases, when you have done a transformation on your dataframe, you have to re-index the rows. For that, you can use the `reset_index()` method. For instance:

```
zoo.sort_values(by = ['water_need'], ascending = False).reset_index()
```

```
In [18]: zoo.sort_values(by = ['water_need'], ascending = False).reset_index()
```

Out[18]:

	index	animal	uniq_id	water_need
0	1	elephant	1002	600
1	16	lion	1017	600
2	2	elephant	1003	550
3	0	elephant	1001	500
4	17	lion	1018	500
5	20	kangaroo	1021	430
6	15	lion	1016	420
7	19	kangaroo	1020	410
8	21	kangaroo	1022	410
9	18	lion	1019	390
10	5	tiger	1006	330
11	4	tiger	1005	320
12	7	tiger	1008	310
13	3	tiger	1004	300
14	6	tiger	1007	290
15	10	zebra	1011	240
16	11	zebra	1012	230
17	9	zebra	1010	220
18	12	zebra	1013	220
19	8	zebra	1009	200
20	13	zebra	1014	100
21	14	zebra	1015	80

Nicer? For sure!

As you can see, our new dataframe kept the old indexes, too. If you want to remove them, just add the `drop = True` parameter:

```
zoo.sort_values(by = ['water_need'], ascending = False).reset_index(drop = True)
```

```
In [19]: zoo.sort_values(by = ['water_need'], ascending = False).reset_index(drop = True)
```

Out[19]:

	animal	uniq_id	water_need
0	elephant	1002	600
1	lion	1017	600
2	elephant	1003	550
3	elephant	1001	500
4	lion	1018	500
5	kangaroo	1021	430
6	lion	1016	420
7	kangaroo	1020	410
8	kangaroo	1022	410
9	lion	1019	390
10	tiger	1006	330
11	tiger	1005	320
12	tiger	1008	310
13	tiger	1004	300
14	tiger	1007	290
15	zebra	1011	240
16	zebra	1012	230
17	zebra	1010	220
18	zebra	1013	220
19	zebra	1009	200
20	zebra	1014	100
21	zebra	1015	80

Fillna

(Note: `fillna` is basically fill + na in one world. If you ask me, it's not the smartest name, but this is what we have.)

Let's rerun the left-merge method that we have used above:

```
zoo.merge(zoo_eats, how = 'left')
```

```
In [12]: zoo.merge(zoo_eats, how = 'left')
```

Out[12]:

	animal	uniq_id	water_need	food
0	elephant	1001	500	vegetables
1	elephant	1002	600	vegetables
2	elephant	1003	550	vegetables
3	tiger	1004	300	meat
4	tiger	1005	320	meat
5	tiger	1006	330	meat
6	tiger	1007	290	meat
7	tiger	1008	310	meat
8	zebra	1009	200	vegetables
9	zebra	1010	220	vegetables
10	zebra	1011	240	vegetables
11	zebra	1012	230	vegetables
12	zebra	1013	220	vegetables
13	zebra	1014	100	vegetables
14	zebra	1015	80	vegetables
15	lion	1016	420	NaN
16	lion	1017	600	NaN
17	lion	1018	500	NaN
18	lion	1019	390	NaN
19	kangaroo	1020	410	vegetables
20	kangaroo	1021	430	vegetables
21	kangaroo	1022	410	vegetables

Remember? These are all our animals. The problem is that we have `NaN` values for lions. `NaN` itself can be really distracting, so I usually like to replace it with something more meaningful. In some cases, this can be a `0` value, or in other cases a specific string value, but this time, I'll go with `unknown`. Let's use the `fillna()` function, which basically finds and replaces all `NaN` values in our dataframe:

```
zoo.merge(zoo_eats, how = 'left').fillna('unknown')
```

```
In [27]: zoo.merge(zoo_eats, how = 'left').fillna('unknown')
```

Out[27]:

	animal	uniq_id	water_need	food
0	elephant	1001	500	vegetables
1	elephant	1002	600	vegetables
2	elephant	1003	550	vegetables
3	tiger	1004	300	meat
4	tiger	1005	320	meat
5	tiger	1006	330	meat
6	tiger	1007	290	meat
7	tiger	1008	310	meat
8	zebra	1009	200	meat
9	zebra	1010	220	meat
10	zebra	1011	240	meat
11	zebra	1012	230	meat
12	zebra	1013	220	meat
13	zebra	1014	100	meat
14	zebra	1015	80	meat
15	lion	1016	420	unknown
16	lion	1017	600	unknown
17	lion	1018	500	unknown
18	lion	1019	390	unknown
19	kangaroo	1020	410	vegetables
20	kangaroo	1021	430	vegetables
21	kangaroo	1022	410	vegetables

Note: since we know that lions eat meat, we could have written `zoo.merge(zoo_eats, how = 'left').fillna('meat')` *, as well.*

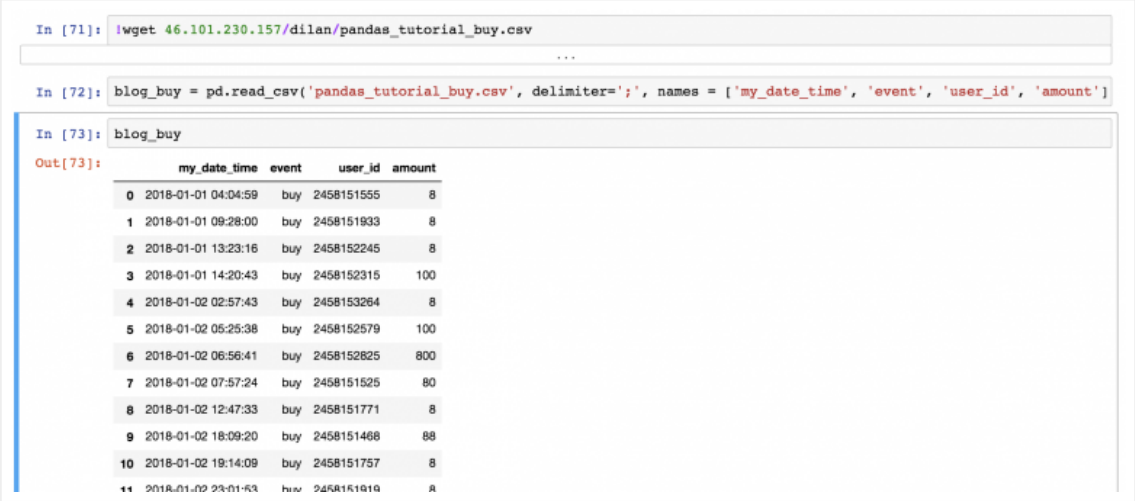
Test yourself

Okay, you've gotten through the article! Great job!
Here's your final test task!

Let's get back to our `article_read` dataset.
(Note: Remember, this dataset holds the data of a travel blog. If you don't have the data yet, you can download it from here. Or you can go through the whole download, open, store process step by step by reading the first episode of this pandas tutorial.)

Download another dataset, too: `blog_buy` . You can do that by running these two lines in your Jupyter Notebook:

```
!wget 46.101.230.157/dilan/pandas_tutorial_buy.csv
blog_buy = pd.read_csv('pandas_tutorial_buy.csv', delimiter=';',
names = ['my_date_time', 'event', 'user_id', 'amount'])
```



```
In [71]: !wget 46.101.230.157/dilan/pandas_tutorial_buy.csv
...
In [72]: blog_buy = pd.read_csv('pandas_tutorial_buy.csv', delimiter=';', names = ['my_date_time', 'event', 'user_id', 'amount'])
In [73]: blog_buy
Out[73]:
```

	my_date_time	event	user_id	amount
0	2018-01-01 04:04:59	buy	2458151555	8
1	2018-01-01 09:28:00	buy	2458151933	8
2	2018-01-01 13:23:16	buy	2458152245	8
3	2018-01-01 14:20:43	buy	2458152315	100
4	2018-01-02 02:57:43	buy	2458153264	8
5	2018-01-02 05:25:38	buy	2458152579	100
6	2018-01-02 06:56:41	buy	2458152825	800
7	2018-01-02 07:57:24	buy	2458151525	80
8	2018-01-02 12:47:33	buy	2458151771	8
9	2018-01-02 18:09:20	buy	2458151468	88
10	2018-01-02 19:14:09	buy	2458151757	8
11	2018-01-02 23:01:53	buy	2458151919	8

The `article_read` dataset shows all the users who read an article on the blog, and the `blog_buy` dataset shows all the users who bought something on the very same blog between `2018-01-01` and `2018-01-07`.

I have two questions for you:

- **TASK #1: What's the average (mean) revenue between `2018-01-01` and `2018-01-07` from the users in the `article_read` dataframe?**
- **TASK #2: Print the top 3 countries by total revenue between `2018-01-01` and `2018-01-07`! (Obviously, this concerns the users in the `article_read` dataframe again.)**

SOLUTION for TASK #1

The average revenue is: `1.0852`

Here's the code:

```
step_1 = article_read.merge(blog_buy, how = 'left', left_on =
'user_id', right_on = 'user_id')
step_2 = step_1.amount
step_3 = step_2.fillna(0)
result = step_3.mean()
result
```

Note: for ease of understanding, I broke this down into "steps" – but you could also bring all these functions into one line.

A short explanation:

- *(On the screenshot, at the beginning, I included the two extra cells where I import pandas and numpy, and where I read the csv files into my Jupyter Notebook.)*
- **In step_1**, I merged the two tables (`article_read` and `blog_buy`) based on the `user_id` columns. I kept all the readers from `article_read` , even if they didn't buy anything, because `0` s should be counted in to the average revenue value. And I removed everyone who bought something but wasn't in the `article_read` dataset (that was fixed in the task). So all in all that led to a *left join*.
- **In step_2,** I removed all the unnecessary columns, and kept only `amount` .
- **In step_3,** I replaced `NaN` values with `0` s.
- And eventually I did the `.mean()` calculation.

SOLUTION for TASK #2

The code is:

```
step_1 = article_read.merge(blog_buy, how = 'left', left_on =
'user_id', right_on = 'user_id')
step_2 = step_1.fillna(0)
step_3 = step_2.groupby('country').sum()
step_4 = step_3.amount
step_5 = step_4.sort_values(ascending = False)
step_5.head(3)
```