Photo by Zoe Nicolaou on Unsplash

9 Useful Pandas Methods You Might Have Not Heard About

They can make your daily work easier and faster.

Eryk Lewinson 3 days ago · 7 min read ★

In this article, I wanted to quickly show a few useful `pandas` methods/functions, which can come in handy during your daily work. To manage expectations, this is not an article showing the basic functionalities of `pandas` and there is no particular theme to the methods. Without further ado, let's start!

1. hasnans

There are many ways of inspecting whether a Series/DataFrame contains missing values, including dedicated libraries such as `missingno`. A simple way to check if a column of a DataFrame contains missing values could look as follows:

```
df["column"].isna().any()
```

Alternatively, we can use the `hasnans` method of a pd.Series for slightly less typing:

```
df["column"].hasnans
```

One downside is that this method does not work for a DataFrame, though we can easily use it in combination with list comprehension:

```
[df[col].hasnans for col in df.columns]
```

Naturally, we lose the names of the columns using this approach. So I would say the primary use case of `hasnans` is doing quick checks on single columns.

2. transform

The `transform` method is used for applying transformations to an entire DataFrame or different transformations to selected columns. However, what makes this method stand out is that it returns an object of the same shape as the input.

That is why `transform` can be a big time-saver when dealing with aggregates. Normally, if we wanted to have some aggregate values (think sum or mean per group) in the original DataFrame, we would create a new, smaller DataFrame using `groupby` and then merge it back to the original DataFrame. That is a bit time-consuming and produces unnecessary intermediate objects.

With `transform`, we can do it in one go! Let's create a DataFrame with some shop IDs and their sales over multiple days (for simplicity, we skip the dates).

```
df = pd.DataFrame(data={"shop_id": [1, 1, 1, 2, 2, 2, 3, 3],
                        "sales": [10, 40, 50, 20, 20, 40, 20, 50]})
df
```

useful_pandas_1.py hosted with ♡ by GitHub view raw

	shop_id	sales
0	1	10
1	1	40
2	1	50
3	2	20
4	2	20
5	2	40
6	3	20
7	3	50

Image by author

We can easily add the sum of sales per shop using the following snippet:

```
df["total_sales_per_shop"] = df.groupby("shop_id")["sales"].transform("sum")
df
```

useful_pandas_2.py hosted with ♡ by GitHub view raw

	shop_id	sales	total_sales_per_shop
0	1	10	100
1	1	40	100
2	1	50	100
3	2	20	80
4	2	20	80
5	2	40	80
6	3	20	70
7	3	50	70

Image by author

`transform` definitely simplifies the process. One more advanced use case of `transform` can be for aggregate filtering. For example, we want to extract rows concerning shops which sold more than 75 units of goods. And ideally without creating a new column in the output DataFrame. We can do so as follows:

```
df = pd.DataFrame(data={"shop_id": [1, 1, 1, 2, 2, 2, 3, 3],
                        "sales": [10, 40, 50, 20, 20, 40, 20, 50]})
df.loc[df.groupby("shop_id")["sales"].transform("sum") > 75]
```

useful_pandas_3.py hosted with ♡ by GitHub view raw

	shop_id	sales
0	1	10
1	1	40
2	1	50
3	2	20
4	2	20
5	2	40

Image by author

3. merge_asof

In principle, this function is similar to a standard left join, however, we match on the nearest key rather than equal keys. This will become clearer with an example.

We also need to know that there are 3 directions of joining using `merge_asof` (the definitions are adapted from the documentation):

- `backward` — for each row in the left DataFrame we select the last row in the right DataFrame whose `on` key is less than or equal to the left's key. This is the default one.
- `forward` — for each row in the left DataFrame we select the first row in the right DataFrame whose `on` key is greater than or equal to the left's key.
- `nearest` — for each row in the left DataFrame we search selects the row in the right DataFrame whose `on` key is closest in absolute distance to the left's key.

Another thing to take into consideration is that both DataFrames must be sorted by the key.

I will not try to reinvent the wheel and will use the example provided in the documentation. Here, we have two DataFrames, one containing trade information and the other one containing quotes. Both contain detailed timestamps (including milliseconds).

```
trades = pd.DataFrame(
        {
            "time": [
                pd.Timestamp("2016-05-25 13:30:00.023"),
                pd.Timestamp("2016-05-25 13:30:00.038"),
                pd.Timestamp("2016-05-25 13:30:00.048"),
                pd.Timestamp("2016-05-25 13:30:00.048"),
                pd.Timestamp("2016-05-25 13:30:00.048")
            ],
            "ticker": ["MSFT", "MSFT", "GOOG", "GOOG", "AAPL"],
            "price": [51.95, 51.95, 720.77, 720.92, 98.0],
            "quantity": [75, 155, 100, 100, 100]
        }
    )

quotes = pd.DataFrame(
    {
        "time": [
            pd.Timestamp("2016-05-25 13:30:00.023"),
            pd.Timestamp("2016-05-25 13:30:00.023"),
            pd.Timestamp("2016-05-25 13:30:00.030"),
            pd.Timestamp("2016-05-25 13:30:00.041"),
            pd.Timestamp("2016-05-25 13:30:00.048"),
            pd.Timestamp("2016-05-25 13:30:00.049"),
            pd.Timestamp("2016-05-25 13:30:00.072"),
            pd.Timestamp("2016-05-25 13:30:00.075")
        ],
        "ticker": [
                "GOOG",
                "MSFT",
```

```
                "MSFT",
                "MSFT",
                "GOOG",
                "AAPL",
                "GOOG",
                "MSFT"
            ],
            "bid": [720.50, 51.95, 51.97, 51.99, 720.50, 97.99, 720.50, 52.01],
            "ask": [720.93, 51.96, 51.98, 52.00, 720.93, 98.01, 720.88, 52.03]
        }
    )
```

useful_pandas_4.py hosted with ♡ by GitHub — view raw

Then, we merge the two DataFrames within `2ms` between the quote time and the trade time:

```
df = pd.merge_asof(
    trades, quotes, on="time", by="ticker", tolerance=pd.Timedelta("2ms")
)
df
```

useful_pandas_5.py hosted with ♡ by GitHub — view raw

What results in the following DataFrame:

	time	ticker	price	quantity	bid	ask
0	2016-05-25 13:30:00.023	MSFT	51.95	75	51.95	51.96
1	2016-05-25 13:30:00.038	MSFT	51.95	155	NaN	NaN
2	2016-05-25 13:30:00.048	GOOG	720.77	100	720.50	720.93
3	2016-05-25 13:30:00.048	GOOG	720.92	100	720.50	720.93
4	2016-05-25 13:30:00.048	AAPL	98.00	100	NaN	NaN

Image by author

It is easiest to understand the logic of the merge by following up a few examples from the output frame back to the two DataFrames.

4. insert

This one is a simple, yet handy method. We can use it to insert a column into a specific location in a DataFrame. The most common use case is when we would like to add a column with extra information for easier analysis of the output. For our and stakeholders' convenience, such information would be most useful as one of the first columns. We can do so with the following snippet:

```
df = pd.DataFrame(data={"a": [1, 2, 3],
                        "b": [4, np.nan, 6]})

df.insert(0, "first", [1, 2, 3])
df
```

useful_pandas_6.py hosted with ♡ by GitHub — view raw

	first	a	b
0	1	1	4.0
1	2	2	NaN
2	3	3	6.0

Image by author

5. explode

explode is useful when we want to expand lists within a DataFrame into multiple rows. Imagine the following case — we have two IDs and multiple values stored within lists for each one of them. We would like to have a row per each ID-value combination.

```
df = pd.DataFrame(data={"id": [1, 2],
                        "values": [[1, 2, 3], [4, 5, 6]]})
df
```

useful_pandas_7.py hosted with ♡ by GitHub view raw

	id	values
0	1	[1, 2, 3]
1	2	[4, 5, 6]

Image by author

```
df.explode("values", ignore_index=True)
```

	id	values
0	1	1
1	1	2
2	1	3
3	2	4
4	2	5
5	2	6

Image by author

6. str

When working with columns containing strings, we can use `str`, which allows us to use a variety of handy methods that can be applied to the entire column in a vectorized manner. For example, we can easily split a column containing full addresses into separate components. First, let's create the DataFrame.

```
df = pd.DataFrame(data={
    "address": ["1st Street, New York, NY 10000, USA",
                "2nd Street, New York, NY 10001, USA"]
})
```

```
5  df
```


	address
0	1st Street, New York, NY 10000, USA
1	2nd Street, New York, NY 10001, USA

Image by author

Then, we can use the `str.split` method to create the new columns.

```
df = df["address"].str.split(",", expand=True)
df.columns = ["street", "city", "zip_code", "country"]
df
```


	street	city	zip_code	country
0	1st Street	New York	NY 10000	USA
1	2nd Street	New York	NY 10001	USA

Image by author

Another possibility is to use the `str.replace` method for modifying the strings.

7. read_clipboard

This one can be especially useful when you frequently work with Excel/Google sheets or receive data in such files. Using `read_clipboard`, we can easily load the data from the source sheet to `pandas` using our computer's clipboard. This way, we can bypass the need to save the file to `.csv` or `.xls`. Naturally, this is mostly useful for some small, ad-hoc analyses.

id	value
1	3
1	5
2	1
2	10

Image by author

Above you can see an example of a simple table in Google sheets. We select the range, press `cmd+c` (or `control+c` for Windows), and then easily load

data to Python using the snippet below:

```
pd.read_clipboard()
```

What returns the following:

	id	value
0	1	3
1	1	5
2	2	1
3	2	10

Image by author

8. eval

The `eval` method uses string expressions to efficiently compute operations on DataFrames. According to the documentation:

> *The following arithmetic operations are supported: `+, -, *, /, **, %, //` (python engine only) along with the following boolean operations: `|` (or), `&` (and), and `~` (not). Additionally, the `'pandas'` parser allows the use of `and`, `or`, and `not` with the same semantics as the corresponding bitwise operators.*

```
df = pd.DataFrame(data={"a": [1, 2, 3],
                        "b": [4, 5, 6]})
df = df.eval("c = a * b")
df
```

useful_pandas_10.py hosted with ♡ by GitHub view raw

	a	b	c
0	1	4	4
1	2	5	10
2	3	6	18

Image by author

9. `squeeze`

Remember the case when you wanted to extract a scalar value from a DataFrame, and after some filtering, you used `.values[0]` to access the final output? And then you probably thought that there must be a nicer way to do it. Indeed there is — the `squeeze` method.

First, let's define a DataFrame:

```
df = pd.DataFrame(data={"a": [1, 2, 3],
```

```
2                               "b": [4, np.nan, 6]})
```

useful_pandas_11.py hosted with ♡ by GitHub view raw

Then, we can try to access a particular value in the DataFrame, for example:

```
df.loc[df["a"] == 3, "b"]
```

And the output looks as follows:

```
2    6.0
Name: b, dtype: float64
```

For a simple `print` this is ok, but it might cause some issues when we want to assign this value to a variable or even place it in another DataFrame.

The elegant solution to get the scalar value is to add the `squeeze` method at the end:

```
df.loc[df["a"] == 3, "b"].squeeze()
```

What returns:

```
6.0
```

Before you go

You can find the code used for this article on my GitHub. Also, any constructive feedback is welcome. I would be very happy to hear about some `pandas` functionalities that make your work easier! You can reach out to me on Twitter or in the comments.

If you liked this article, you might also be interested in one of the following:

Make working with large DataFrames easier, at least for your memory

Reduce the size of your DataFrames by up to 90%!

towardsdatascience.com

Explaining the SettingWithCopyWarning in pandas

If you are wondering what causes the SettingwithCopyWarning, this is the place for you!

is the place for you.

towardsdatascience.com

Embed your interactive visualizations in Medium posts using datapane

towardsdatascience.com

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

You'll need to sign in or create an account to receive this newsletter.

Education Data Science Python Pandas Data Wrangling

About Help Legal