✦ Member-only story

Manipulating Values in Polars DataFrames

Learn how to use the `various` methods in Polars to manipulate your dataframes

Wei-Meng Lee · Follow
Published in Towards Data Science · 6 min read · Jul 20, 2022

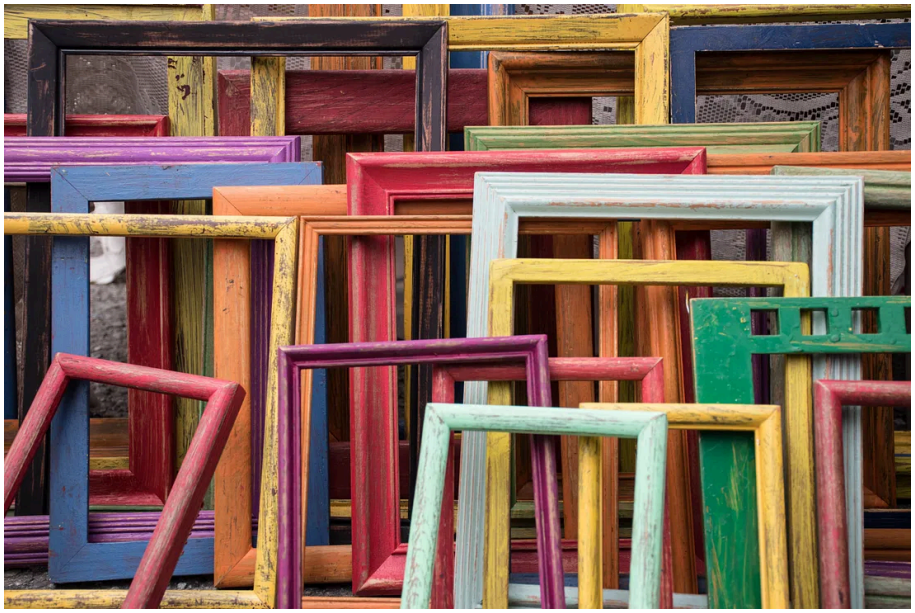


Photo by Jessica Ruscello on Unsplash

Up till this point, I have talked about how to use the Polars DataFrames, and why it is a better DataFrame library compared to Pandas. Continuing with our exploration of Polars, in this article I will show you how to manipulate your Polars DataFrame, specifically:

- How to change the values for each column/row
- How to sum up the values for each column/row
- How to add a new column/row to the existing dataframe

Ready? Let's go!

Creating the Sample DataFrame

Let's create a Polars DataFrame using a list of tuples:

```
import polars as pl

matrix = [
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 9),
    (10, 11, 12),
    (13, 14, 15),
    (16, 17, 18)
]

df = pl.DataFrame(matrix, columns=list('abc'))
df
```

The dataframe looks like this:

shape: (6, 3)

a	b	c
i64	i64	i64
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18

Image by author

Let's examine some of the methods that you can call to manipulate the values in the dataframe.

Using the apply() Method

The `apply()` method can used on:

- an individual column in a dataframe, or
- an entire dataframe

Applying on columns

For example, say you want to multiply all the values in the '**a**' column by 2. You can do the following:

```
df.select(
    pl.col('a').apply(lambda x: x*2)
)
```

All the values in column '**a**' will now be multiplied by 2:

shape: (6, 1)

a
i64
2
8
14
20
26
32

Image by author

In the lambda function above, `x` will take on the individual values of the column **a**. When applied to a column, the `apply()` method sends in the values for a column one-by-one. This provides the opportunity for you to examine each value before deciding how you want to change the values. For example, you can multiply only those values which are greater or equal to 5:

```
df.select(
    pl.col('a').apply(lambda x: x*2 if x>=5 else x)
)
```

This will produce the following output:

shape: (6, 1)

a
i64
1
4
14
20
26
32

Image by author

> *In general, implementing logic using the `apply()` method is slower and more memory intensive than implementing your logic using expressions. This is because expressions can be parallelized and optimized, and the logic implemented in an expression is implemented in Rust, which is faster than its implementation in Python (implemented in a lambda function, for example). So, whenever possible, use expressions instead of using the `apply()` function. As an example, the earlier `apply()` method can also be rewritten using an expression:*

```
pl.col('a').apply(lambda x: x*2)
# rewritten as an expression
```

```
    pl.col('a') * 2
```

Notice that the result only contains a single column. If you want the rest of the columns to be in the result as well, use the `select()` and `exclude()` methods:

```
q = (
    df
    .lazy()
    .select(
        [
            pl.col('a').apply(lambda x: x*2),
            pl.exclude('a')
        ]
    )
)

q.collect()
```

Now the result contains all the columns:

shape: (6, 3)

a	b	c
i64	i64	i64
2	2	3
8	5	6
14	8	9
20	11	12
26	14	15
32	17	18

Image by author

If you want to multiply all columns by 2, select all columns using `pl.col('*')`:

```
q = (
    df
    .lazy()
    .select(
        pl.col('*').apply(lambda x: x*2)
    )
)

q.collect()
```

All the columns would now be multiplied by 2:

shape: (6, 3)

a	b	c
i64	i64	i64
2	4	6
8	10	12
14	16	18
20	22	24
26	28	30
32	34	36

Image by author

If you want to multiply column '**a**' by 2 and then store the result as another column, use the `alias()` method:

```
q = (
    df
    .lazy()
    .select(
        [
            pl.col('*'),
            pl.col('a').apply(lambda x: x*2).alias('x*2'),
        ]
    )
)

q.collect()
```

The result would now have an additional column:

shape: (6, 4)

a	b	c	x*2
i64	i64	i64	i64
1	2	3	2
4	5	6	8
7	8	9	14
10	11	12	20
13	14	15	26
16	17	18	32

Image by author

Using the map() method

Another function that is similar to the `apply()` method is the `map()` method. Unlike the `apply()` method, the `map()` method sends in the values of a column as a single Polars Series:

```
df.select(
    pl.col('a').map(lambda x: x*2)
```

```
)
```

In the lambda function above, `x` is a Polars Series containing the values of the column **a**. The above statement produces the following output:

shape: (6, 1)

a
i64
2
8
14
20
26
32

Image by author

Applying on rows

Observe that so far the `apply()` method is applied to *columns* in a dataframe. What if you want to apply to *rows* in a dataframe? In this case, call the `apply()` method on the dataframe directly.

To understand how it works, I wrote a `test` function to print out the value that it gets when the `apply()` function is applied to the dataframe:

```
def test(x):
    print(x)
    return x

df.apply(test)
```

It returns the following:

```
(1, 2, 3)
(4, 5, 6)
(7, 8, 9)
(10, 11, 12)
(13, 14, 15)
(16, 17, 18)
```

This means that the `apply()` function, when applied to a dataframe, sends the values of each row as a tuple to the receiving function. This is useful for some use cases. For example, say you need to perform an integer division of all the numbers in a row by 2 if the sum of them is greater than 10, then you can write the lambda function as:

```
df.apply(lambda x: tuple([i // 2 for i in x]) if sum(x) > 10 else x)
```

And the result will look like this:

shape: (6, 3)

column_0	column_1	column_2
i64	i64	i64
1	2	3
2	2	3
3	4	4
5	5	6
6	7	7
8	8	9

Image by author

If you want to duplicate all the columns in the dataframe, you can also use the `apply()` method:

```
df.apply(lambda x: x*2)
```

The dataframe now has six columns:

column_0	column_1	column_2	column_3	column_4	column_5
i64	i64	i64	i64	i64	i64
1	2	3	1	2	3
4	5	6	4	5	6
7	8	9	7	8	9
10	11	12	10	11	12
13	14	15	13	14	15
16	17	18	16	17	18

Image by author

> *Note that the `apply()` function cannot be applied to a LazyFrame.*

Summing up values in the DataFrame

Often, you need to sum up all the values in your dataframe either row-wise, or column-wise.

By column

The easiest way to sum up the values for each column is to use the `sum()` method on the dataframe:

```
df.sum()
```

shape: (1, 3)

a	b	c
i64	i64	i64
51	57	63

Image by author

To append the result above to the existing dataframe, use the `concat()` method:

```
pl.concat([df, df.sum()])
```

shape: (7, 3)

a	b	c
i64	i64	i64
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
51	57	63

Image by author

By row

To sum up the values of all the columns for each row, use the `sum()` method with the `axis` parameter set to `1`:

```
df.sum(axis=1)
```

The result is a *Polars Series*:

```
shape: (6,)
Series: 'a' [i64]
[
        12
        30
        48
        66
        84
        102
]
```

Image by author

> *Think of a Polars Series as a single column in a dataframe*

You can also use the `select()` method to select the columns that you want to sum up:

```
df.select(pl.col('*')).sum(axis=1)
```

The following code snippet adds the series to the dataframe as a new column:

```
df['sum'] = df.select(pl.col('*')).sum(axis=1)
df
```

shape: (6, 4)

a	b	c	sum
i64	i64	i64	i64
1	2	3	12
4	5	6	30
7	8	9	48
10	11	12	66
13	14	15	84
16	17	18	102

Image by author

If you do not want to use *square bracket indexing* (which is not recommended in Polars), use the `select()` method instead:

```
df.select(
    [
        pl.col('*'),
        df.select(pl.col('*')).sum(axis=1).alias('sum')
    ]
)
```


> *I will be running a workshop on Polars in the upcoming ML Conference (22–24 Nov 2022) in Singapore. If you want a jumpstart on the Polars DataFrame, register for my workshop at https://mlconference.ai/machine-learning-advanced-development/using-polars-for-data-analytics-workshop/.*

Summary

I hope this article added some ammunition to your arsenal for working with your Polars DataFrames. Here is a quick summary of when to use the `apply()` and `map()` methods:

- Call the `apply()` method on an expression to apply a function to *individual* values in a *column(s)* in a dataframe.
- Call the `map()` function on an expression to apply a function to a *column(s)* as a Series in a dataframe.
- Call the `apply()` method on a dataframe to apply a function to *row*s in a dataframe.

Save this article and use it as a quick reference the next time you work with your Polars DataFrame!

Polars Dataframe Map Apply Series

Written by Wei-Meng Lee

Follow

2.8K Followers · Writer for Towards Data Science

ACLP Certified Trainer | Blockchain, Smart Contract, Data Analytics, Machine Learning, Deep Learning, and all things tech (http://calendar.learn2develop.net).

More from Wei-Meng Lee and Towards Data Science

Wei-Meng Lee in AI Advances

Accelerating Hugging Face Pre-trained Models on Apple Silicon...

A Guide to Training and Serving Models Efficiently on M1, M2, and M3 Chips

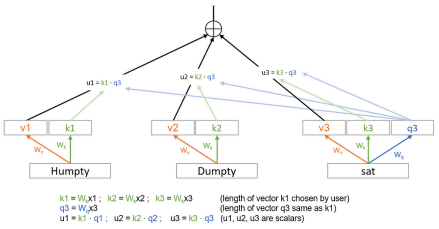
Oct 13 286 3

Shaw Talebi in Towards Data Science

5 AI Projects You Can Build This Weekend (with Python)

From beginner-friendly to advanced

Oct 9 3.5K 60



Rohit Patel in Towards Data Science

Understanding LLMs from Scratch Using Middle School Math

In this article, we talk about how LLMs work, from scratch—assuming only that you know...

Oct 19 1.8K 19

Wei-Meng Lee in AI Advances

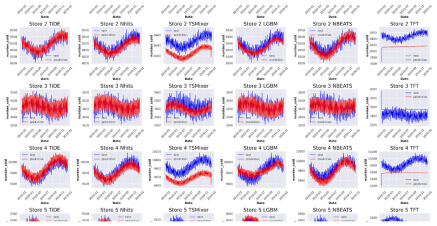
Understanding Tokenization in NLP

Breaking Down Word-Level, Subword-Level, Padding, and Attention Masks for Effective...

4d ago 321 2

See all from Wei-Meng Lee | See all from Towards Data Science

Recommended from Medium



Esther Cifuentes

Comparing Time Series Algorithms

Evaluating Leading Time Series Algorithm with Darts.

Oct 16 83

Tim Zhao

A Better Way to Use the Pandas DataFrame: Treat Each Row as a...

Pandas is a powerful and flexible data manipulation library that is a staple in the...

Aug 13

Lists

Icon Design
36 stories · 436 saves

Staff Picks
753 stories · 1414 saves

Sean Ma in Python in Plain English

Read CSV Files 10x to 40x Faster Using pyarrow and polars

Learning faster Python fast with Sean

Jun 13 21

Thomas Reid in AI Advances

Where's the read_excel() function in DuckDB?

Spoiler Alert: There isn't one. It just does Excel a bit differently.

Oct 23 140 1