Introduction to Pandas apply, applymap and map

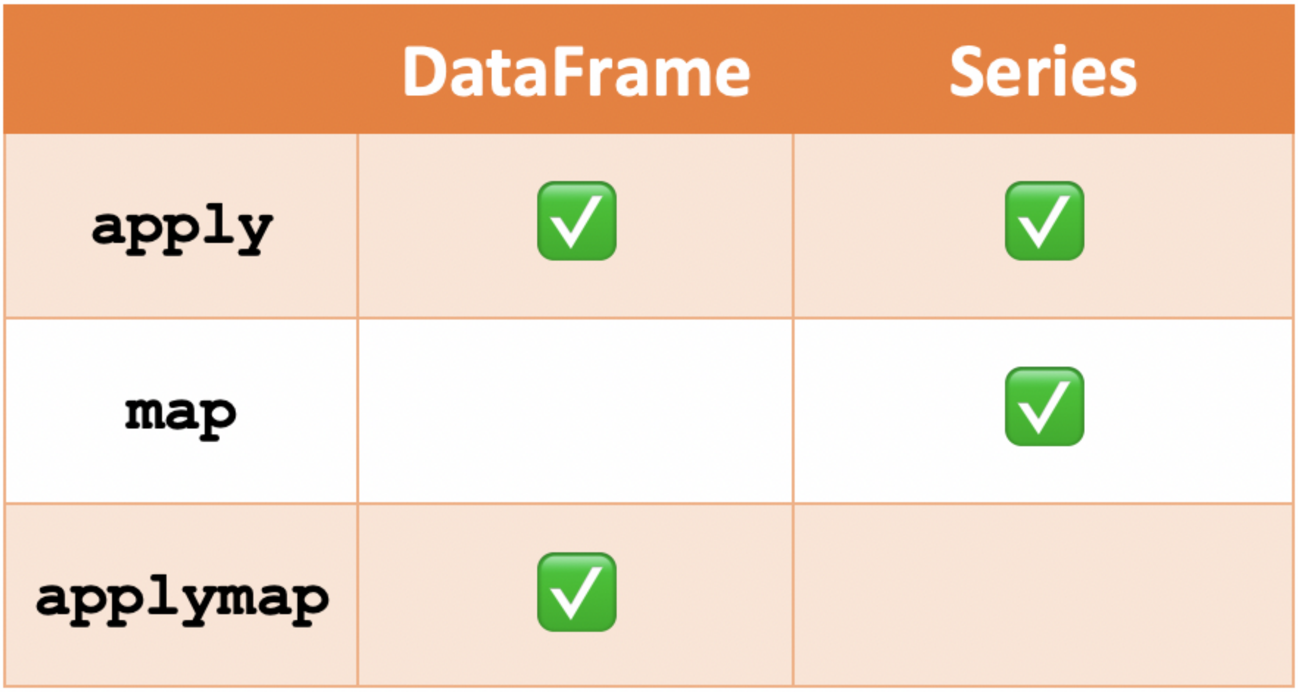
An intuitive Pandas tutorial for how to apply a function using apply() and applymap(), and how to substitute value using map()

Introduction to Pandas apply(), applymap() and map()

In Data Processing, it is often necessary to perform operations (such as statistical calculations, splitting, or substituting value) on a certain row or column to obtain new data. Writing a for-loop to iterate through Pandas DataFrame and Series will do the job, but that doesn't seem like a good idea. The for-loop tends to have more lines of code, less code readability, and slower performance.

Fortunately, there are already great methods that are built into Pandas to help you accomplish the goals! In this article, we will see how to perform operations using `apply()` and `applymap()`, and how to substitute value using `map()`.

First of all, you should be aware that DataFrame and Series will have some or all of these three methods, as follows:



	DataFrame	Series
apply	✅	✅
map		✅
applymap	✅	

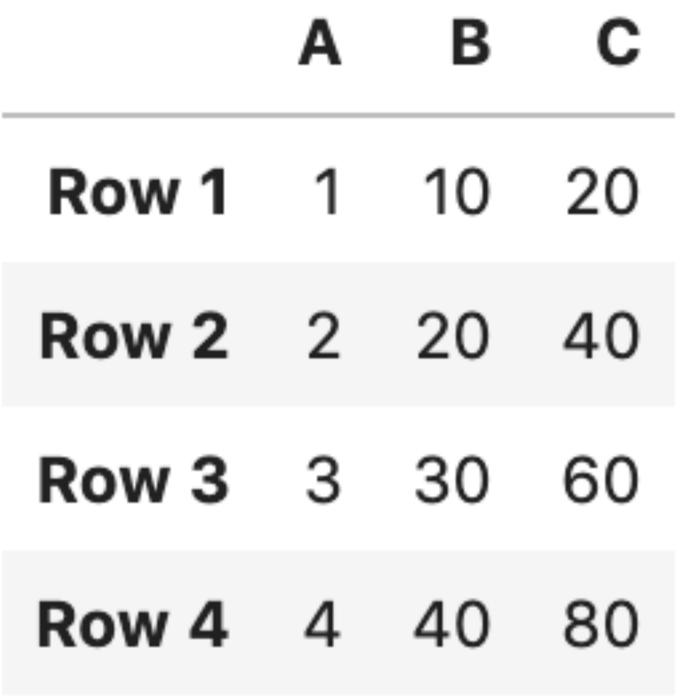
And the Pandas official API reference suggests that:

- `apply()` is used to apply a function **along an axis of the DataFrame** or **on values of Series**.
- `applymap()` is used to apply a function to a DataFrame elementwise.
- `map()` is used to substitute each value in a Series with another value.

Dataset for demonstration

Before we diving into the details, let’s first create a DataFrame for demonstration.

```
import pandas as pddf = pd.DataFrame({ 'A': [1,2,
                   'B': [10,20,30,40],
                   'C': [20,40,60,80]
                  },
                  index=['Row 1', 'Row 2', 'Row 
```



	A	B	C
Row 1	1	10	20
Row 2	2	20	40
Row 3	3	30	60
Row 4	4	40	80

Dataset for apply, applymap and map demonstration

How to use apply()?

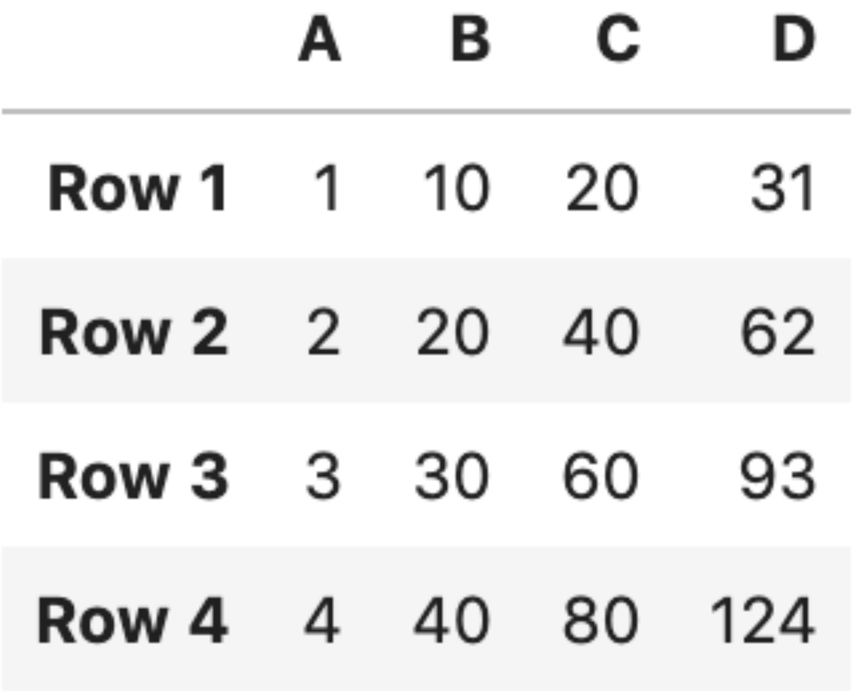
The Pandas `apply()` is used to apply a function **along an axis of the DataFrame** or **on values of Series**.

Let’s begin with a simple example, to sum each row and save the result to a new column “D”

```
# Let's call this "custom_sum" as "sum" is a bui
```

```
def custom_sum(row):
    return row.sum()df['D'] = df.apply(custom_sur
```

And here is the output



	A	B	C	D
Row 1	1	10	20	31
Row 2	2	20	40	62
Row 3	3	30	60	93
Row 4	4	40	80	124

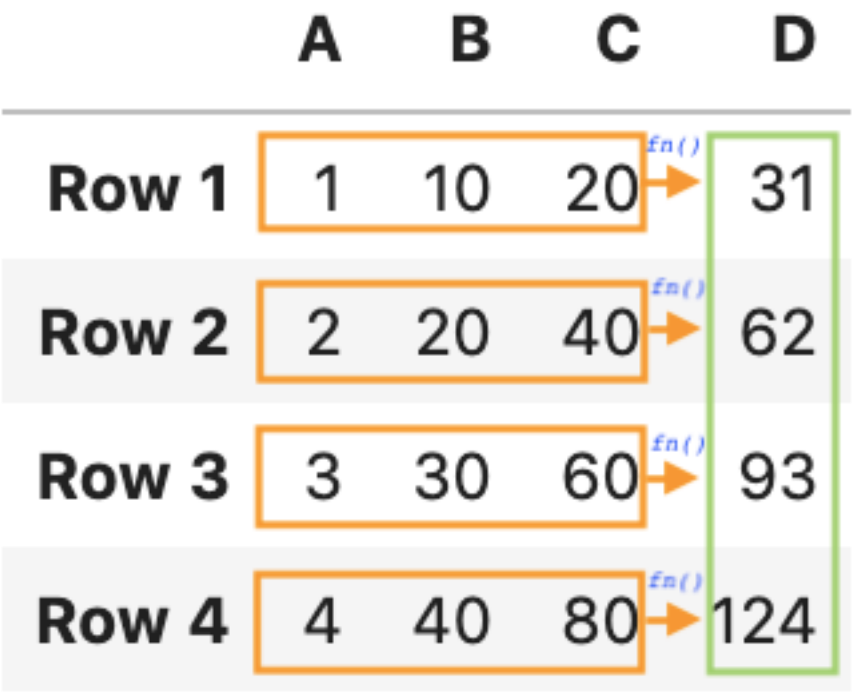
result of df[**'D'**] = df.apply(**custom_sum**, **axis=1**)

Do you really understand what just happened?

Let's take a look `df.apply(`**`custom_sum, axis=1`**`)`

- The first parameter **`custom_sum`** is a function.
- The second parameter `axis` is to specify which axis the function is applied to. `0` for applying the function to each column and `1` for applying the function to each row.

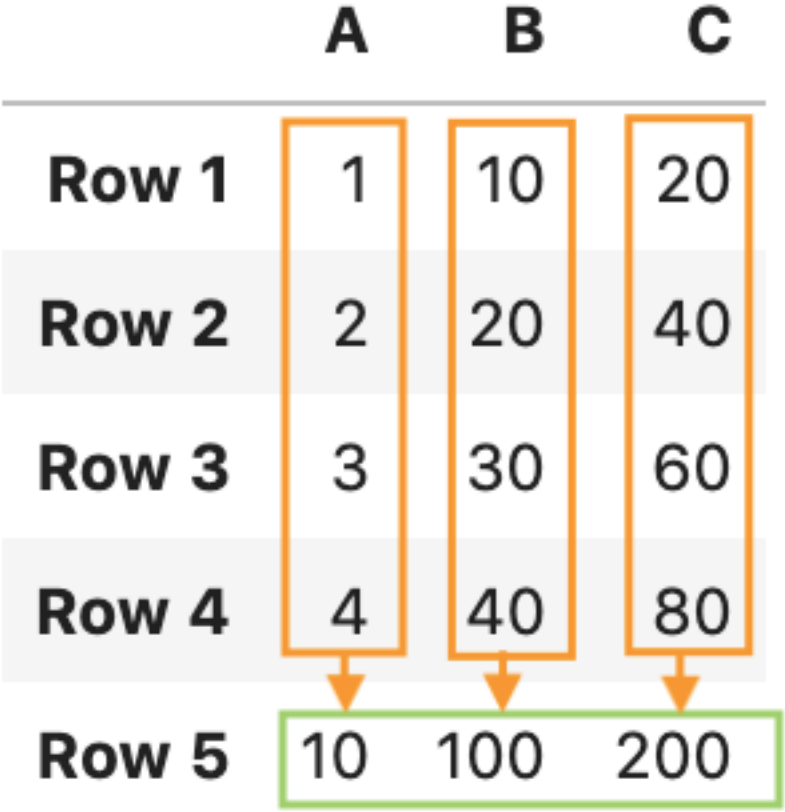
Let me explain this process in a more intuitive way. The second parameter `axis = 1` tells Pandas to use the row. So, the `custom_sum` is applied to each row and returns a new Series with the output of each row as value.



	A	B	C	D
Row 1	1	10	20	31
Row 2	2	20	40	62
Row 3	3	30	60	93
Row 4	4	40	80	124

With the understanding of the sum of each row, the sum of each column is just to use `axis = 0` instead

```
df.loc['Row 5'] = df.apply(custom_sum, axis=0)
```

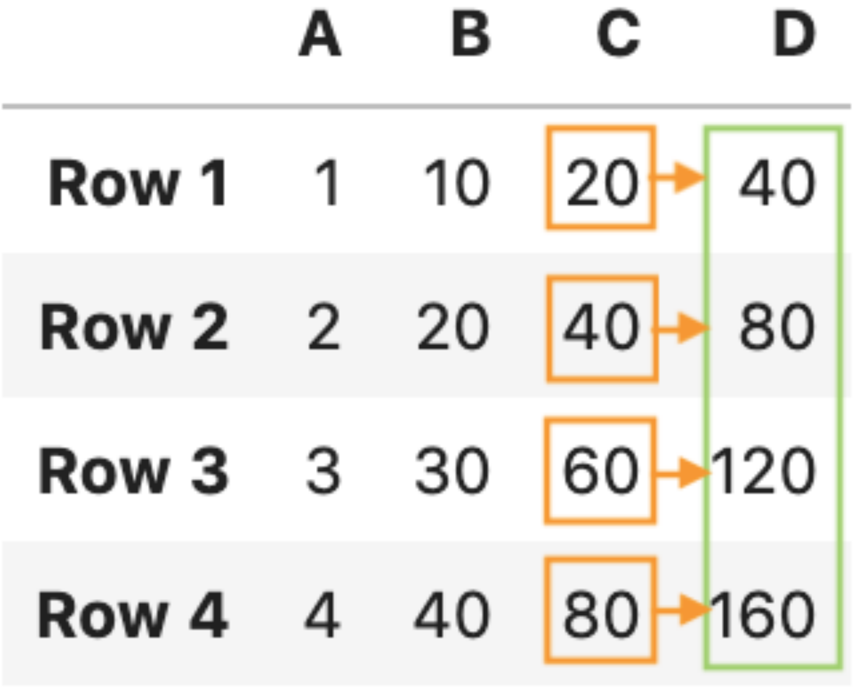


	A	B	C
Row 1	1	10	20
Row 2	2	20	40
Row 3	3	30	60
Row 4	4	40	80
Row 5	10	100	200

So far, we have been talking about `apply()` on a DataFrame. Similarly, `apply()` can be used on the values of Series. For example, multiply the column “C” by 2 and save the result to a new column “D”

```
def multiply_by_2(val):
    return val * 2df['D'] = df['C'].apply(multipl
```

Notice that `df[‘C’]` is used to select the column “C” and then call `apply()` with the only parameter `multiply_by_2`. We don’t need to specify axis anymore because Series is a **one-dimensional array**. The return value is a Series and get assigned to the new column D by `df[‘D’]`.



	A	B	C	D
Row 1	1	10	20	40
Row 2	2	20	40	80
Row 3	3	30	60	120
Row 4	4	40	80	160

output of apply() on a series

Use lambda with apply

You can also use lambda expression with Pandas `apply()` function.

The lambda equivalent for the sum of each row of a DataFrame:

```
df['D'] = df.apply(lambda x:x.sum(), axis=1)
```

The lambda equivalent for the sum of each column of a DataFrame:

```
df.loc['Row 5'] = df.apply(lambda x:x.sum(), axi
```

And the lambda equivalent for multiply by 2 on a Series:

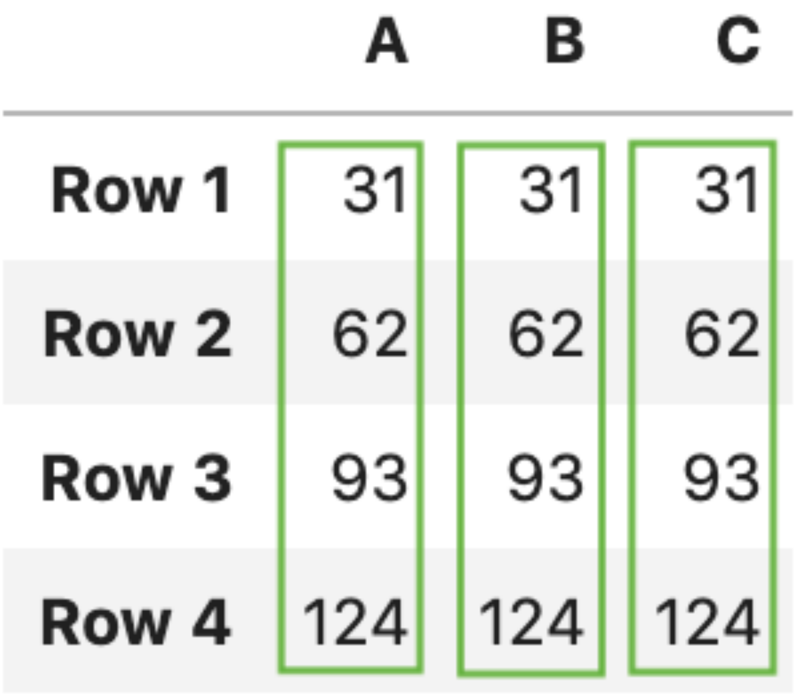
```
df['D'] = df['C'].apply(lambda x:x*2)
```

With result_type parameter

`result_type` is a parameter in `apply()` set to `'expand'`, `'reduce'`, or `'broadcast'` to get the desired type of result.

In the above scenario if `result_type` is set to `'broadcast'` then the output will be a DataFrame substituted by the `custom_sum` value.

```
df.apply(custom_sum, axis=1, result_type='broadca
```



	A	B	C
Row 1	31	31	31
Row 2	62	62	62
Row 3	93	93	93
Row 4	124	124	124

The result is broadcasted to the original shape of the frame, the original index and columns are retained.

To understand `result_type` as `'expand'` and `'reduce'`, we will first create a function that returns a list.

```
def cal_multi_col(row):
    return [row['A'] * 2, row['B'] * 3]
```

Now apply this function across the DataFrame column with `result_type` as `'expand'`

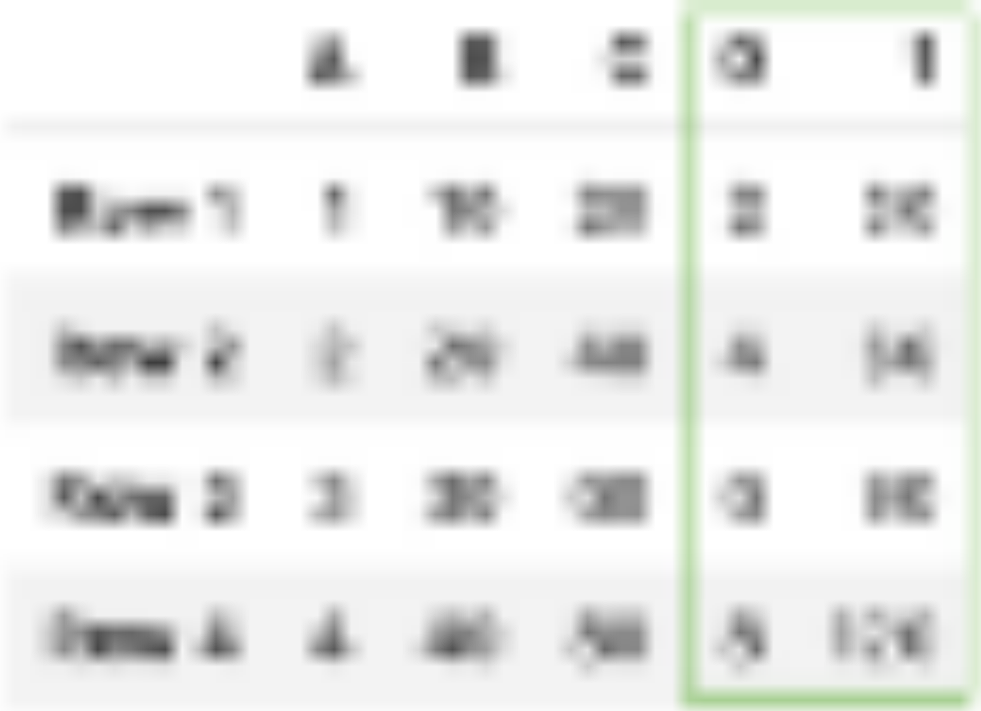
```
df.apply(cal_multi_col, axis=1, result_type='expa
```

The output is a new DataFrame with column names **0** and **1**.

In order to append this to the existing DataFrame, the result has to be kept in a variable so the column names can be accessed by `res.columns`.

```
res = df.apply(cal_multi_col, axis=1, result_type
df[res.columns] = res
```

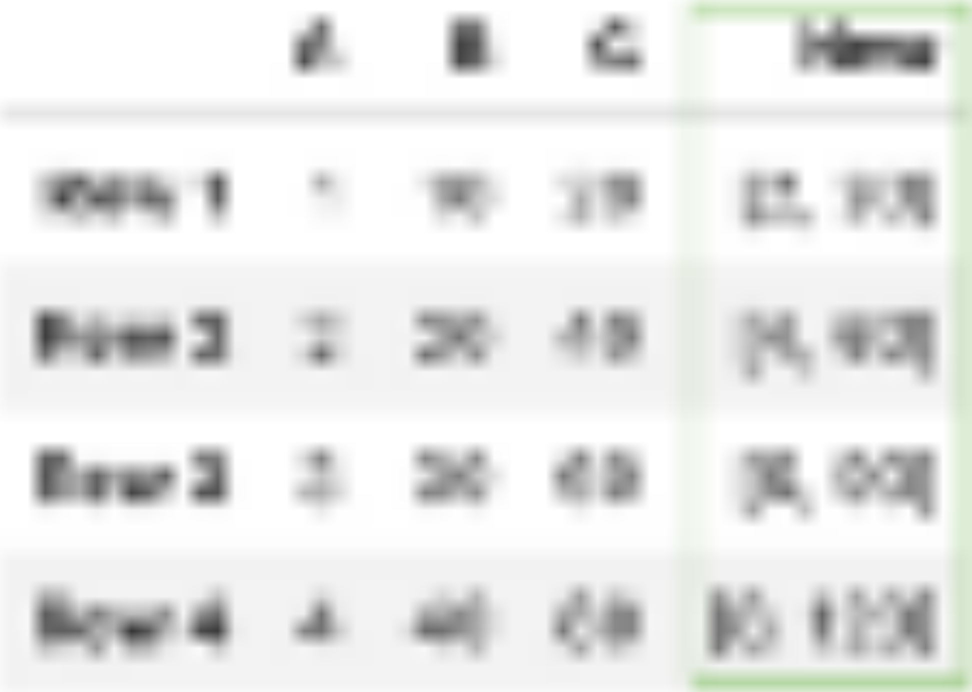
And the output is:



Output of result_type='expand'

Next, apply the function across the DataFrame column with `result_type` as `'reduce'`. `result_type='reduce'` is just opposite of `'expand'` and returns a Series if possible rather than expanding list-like results.

```
df['New'] = df.apply(cal_multi_col, axis=1, resul
```



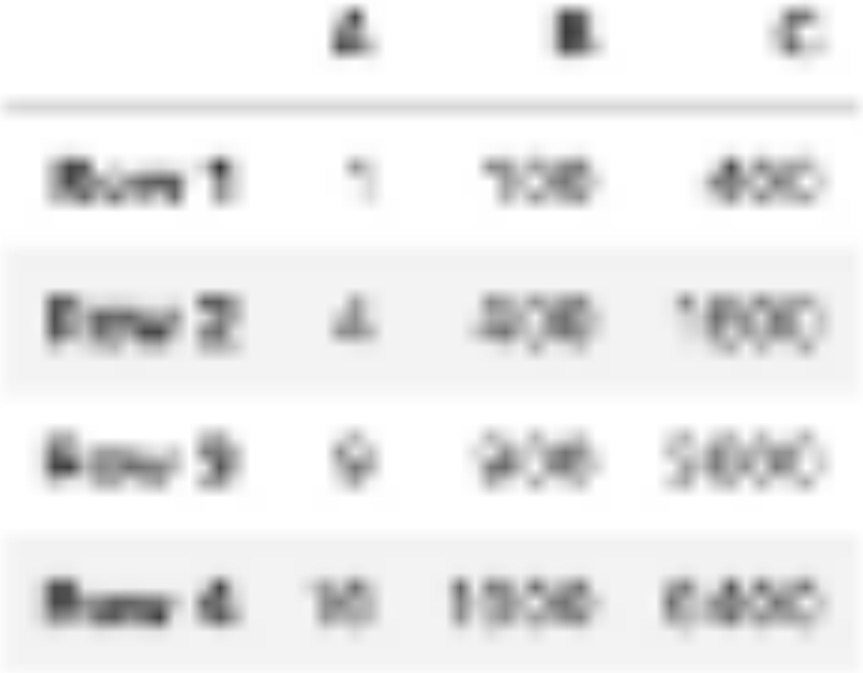
output of result_type='reduce'

How to use applymap()?

`applymap()` is only available in DataFrame and used for element-wise operation across the whole DataFrame. It has been optimized and some cases work much faster than `apply()` , but it's good to compare it with `apply()` before going for any heavier operation.

For example: to output a DataFrame with number squared

```
df.applymap(np.square)
```



	A	B	C
Row 1	1	100	400
Row 2	4	400	1600
Row 3	9	900	3600
Row 4	16	1600	6400

How to use map()?

`map()` is only available in Series and used for substituting each value in a Series with another value. To understand how the `map()` works, we first create a Series.

```
>>> s = pd.Series(['cat', 'dog', np.nan, 'rabbit
>>> s
0      cat
1      dog
2      NaN
3   rabbit
dtype: object
```

`map()` accepts a `dict` or a `Series`. Values that are not found in the `dict` are converted to `NaN`, unless the `dict` has a default value (e.g. `defaultdict`):

```
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})
0   kitten
1    puppy
2      NaN
3      NaN
dtype: object
```

It also accepts a function:

```
>>> s.map('I am a {}'.format)
0       I am a cat
1       I am a dog
2       I am a nan
3    I am a rabbit
dtype: object
```

To avoid applying the function to missing values (and keep them as `NaN`) `na_action='ignore'` can be used:

```
>>> s.map('I am a {}'.format, na_action='ignore')
0     I am a cat
1     I am a dog
2            NaN
3  I am a rabbit
dtype: object
```

Summary

Finally, here is a summary:

For DataFrame:

- `apply()`: It is used when you want to apply a function along the row or column. `axis = 0` for column and `axis = 1` for row.
- `applymap()`: It is used for element-wise operation across the whole DataFrame.

For Series:

- `apply()`: It is used when you want to apply a function on the values of Series.
- `map()`: It is used to substitute each value with another value.

You may be interested in some of my other Pandas articles:

- Pandas concat() tricks you should know
- How to do a Custom Sort on Pandas DataFrame
- When to use Pandas transform() function
- Using Pandas method chaining to improve code readability
- Working with datetime in Pandas DataFrame
- Working with missing values in Pandas
- Pandas read_csv() tricks you should know
- 4 tricks you should know to parse date columns with Pandas read_csv()

More can be found from my Github

Enjoy!

That’s about it. Thanks for reading.