In [1]:

```
import os

from IPython.display import display, HTML
import numpy as np

import pandas as pd
if 'dirty' in pd.__version__:
    raise ValueError(f'Running on dirty branch: {pd.__version__}')
pd.__version__
```

Out[1]:

```
'1.4.0.dev0+727.g4acdb55383'
```

In [2]:

```
css = """
.output {
    flex-direction: row;
}
"""

HTML('<style>{}</style>'.format(css))
```

Out[2]:

In [3]:

```
def compare(line):
    with pd.option_context('new_udf_methods', False):
        try:
            result1 = eval(line)
            if isinstance(result1, pd.DataFrame):
                result1.index.name = 'master'
            elif isinstance(result1, pd.Series):
                result1.name = 'master'
        except Exception as err:
            result1 = f"master failed: {err}"
    with pd.option_context('new_udf_methods', True):
        try:
            result2 = eval(line)
            if isinstance(result2, pd.DataFrame):
                result2.index.name = 'feature'
            elif isinstance(result2, pd.Series):
                result2.name = 'feature'
        except Exception as err:
            result2 = f"feature failed: {err}"
    display(result1, result2)
```

`df.agg(["sum"])` on master will transpose the data when compared to `df.sum()` , whereas feat keeps the same shape (but returns a DataFrame instead of a Series)

In [4]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
compare('df.sum()')
```

```
a     6
b    15
Name: master, dtype: int64
a     6
b    15
Name: feature, dtype: int64
```

In [5]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
compare('df.agg(["sum"])')
```

	a	b
master		
sum	6	15

	sum
feature	
a	6
b	15

This lack of transposing means dtypes are better preserved in the case of different operations.

In [6]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
compare('df.agg(["sum", "mean"])')
```

	a	b
master		
sum	6.0	15.0
mean	2.0	5.0

	sum	mean
feature		
a	6	2.0
b	15	5.0

If the inputs have different dtypes, then the resulting dtypes on master is better preserved.

In [7]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4.0, 5.0, 6.]})
compare('df.agg(["sum", "max"])')
```

	a	b
master		
sum	6	15.0
max	3	6.0

	sum	max
feature		
a	6.0	3.0
b	15.0	6.0

The internals no longer breakup a DataFrame into columns, resulting in better performance on wide data.

In [8]:

```
df = pd.DataFrame({e: range(10000) for e in range(500)})
with pd.option_context('new_udf_methods', False):
    print('master')
    %timeit df.agg(['sum'])
```

```
with pd.option_context('new_udf_methods', True):
    print('feature')
    %timeit df.agg(['sum'])
```

```
master
267 ms ± 4.17 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
feature
8.99 ms ± 120 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This improvement is small on long data.

In [9]:

```
df = pd.DataFrame({e: range(1000000) for e in range(1)})
with pd.option_context('new_udf_methods', False):
    print('master')
    %timeit df.agg(['sum'])
with pd.option_context('new_udf_methods', True):
    print('feature')
    %timeit df.agg(['sum'])
```

```
master
1.5 ms ± 9.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
feature
1.31 ms ± 15.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The difference is also less pronounced when using the ArrayManager, but still significant on wide data

In [10]:

```
with pd.option_context('data_manager', 'array'):
    df = pd.DataFrame({e: range(10000) for e in range(500)})
    with pd.option_context('new_udf_methods', False):
        print('master')
        %timeit df.agg(['sum'])
    with pd.option_context('new_udf_methods', True):
        print('feature')
        %timeit df.agg(['sum'])
```

```
master
263 ms ± 4.75 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
feature
30.4 ms ± 801 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

By not breaking the DataFrame up into columns, we need to swap the column levels when using apply with a transformer

In [11]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
compare('df.apply([np.sqrt])')
```

	a	b
	sqrt	**sqrt**
master		
0	1.000000	2.000000
1	1.414214	2.236068
2	1.732051	2.449490

		sqrt
	a	**b**
feature		

	sqrt	
	a	b
feature		
0	1.000000	2.000000
1	1.414214	2.236068
2	1.732051	2.449490

Swapping of the levels carries over to groupby, and results in a more consistent shape

In [12]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
compare('df.groupby("a").agg("sum")')
```

	b	c
master		
1	4	7
2	5	8
3	6	9

	b	c
feature		
1	4	7
2	5	8
3	6	9

In [13]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
compare('df.groupby("a").agg(["sum", "mean"])')
```

	b		c	
	sum	mean	sum	mean
master				
1	4	4.0	7	7.0
2	5	5.0	8	8.0
3	6	6.0	9	9.0

	sum		mean	
	b	c	b	c
feature				
1	4	7	4.0	7.0
2	5	8	5.0	8.0
3	6	9	6.0	9.0

Swapping of the levels does not happen with dictionary arguments

In [14]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
```

```
compare('df.groupby("a").agg({"b": ["sum", "mean"], "c": ["sum", "mean"]})')
```

	b		c	
	sum	mean	sum	mean
master				
1	4	4.0	7	7.0
2	5	5.0	8	8.0
3	6	6.0	9	9.0

	b		c	
	sum	mean	sum	mean
feature				
1	4	4.0	7	7.0
2	5	5.0	8	8.0
3	6	6.0	9	9.0

apply no longer uses agg for lists or dictionaries giving more consistent results

In [15]:

```
ser = pd.Series([1, 2, 3])
compare('ser.apply(lambda x: x.sum())')
```

```
"master failed: 'int' object has no attribute 'sum'"
"feature failed: 'int' object has no attribute 'sum'"
```

In [16]:

```
ser = pd.Series([1, 2, 3])
compare('ser.apply([lambda x: x.sum()])')
```

```
<lambda>    6
Name: master, dtype: int64
"feature failed: 'int' object has no attribute 'sum'"
```

An odd case noticed due to the test `test_pivot_margins_name_unicode`

In [17]:

```
df = pd.DataFrame({'a': [1, 1, 2]})
compare('df.groupby("a").agg(len)')
```

master
1
2

```
a
1    2
2    1
Name: feature, dtype: int64
```

In [18]:

```
df = pd.DataFrame({'a': [1, 1, 2]})
compare('df.groupby("a").agg([len])')
```

```
'master failed: no results'
```

	len

	len
feature	
1	2
2	1