Performance of Pandas apply vs np.vectorize to create new column from existing columns

Asked 3 years, 7 months ago Modified 1 year, 7 months ago Viewed 61k times

118

75

I am using Pandas dataframes and want to create a new column as a function of existing columns. I have not seen a good discussion of the speed difference between `df.apply()` and `np.vectorize()` , so I thought I would ask here.

The Pandas `apply()` function is slow. From what I measured (shown below in some experiments), using `np.vectorize()` is 25x faster (or more) than using the DataFrame function `apply()` , at least on my 2016 MacBook Pro. **Is this an expected result, and why?**

For example, suppose I have the following dataframe with `N` rows:

```
N = 10
A_list = np.random.randint(1, 100, N)
B_list = np.random.randint(1, 100, N)
df = pd.DataFrame({'A': A_list, 'B': B_list})
df.head()
#     A   B
# 0  78  50
# 1  23  91
# 2  55  62
# 3  82  64
# 4  99  80
```

Suppose further that I want to create a new column as a function of the two columns `A` and `B` . In the example below, I'll use a simple function `divide()` . To apply the function, I can use either `df.apply()` or `np.vectorize()` :

```
def divide(a, b):
    if b == 0:
        return 0.0
    return float(a)/b

df['result'] = df.apply(lambda row: divide(row['A'], row['B']), axis=1)

df['result2'] = np.vectorize(divide)(df['A'], df['B'])

df.head()
#     A   B    result   result2
# 0  78  50  1.560000  1.560000
# 1  23  91  0.252747  0.252747
# 2  55  62  0.887097  0.887097
# 3  82  64  1.281250  1.281250
# 4  99  80  1.237500  1.237500
```

If I increase `N` to real-world sizes like 1 million or more, then I observe that `np.vectorize()` is 25x faster or more than `df.apply()` .

Below is some complete benchmarking code:

```
import pandas as pd
import numpy as np
```

```
import numpy as np
import time

def divide(a, b):
    if b == 0:
        return 0.0
    return float(a)/b

for N in [1000, 10000, 100000, 1000000, 10000000]:

    print ''
    A_list = np.random.randint(1, 100, N)
    B_list = np.random.randint(1, 100, N)
    df = pd.DataFrame({'A': A_list, 'B': B_list})

    start_epoch_sec = int(time.time())
    df['result'] = df.apply(lambda row: divide(row['A'], row['B']), axis=1)
    end_epoch_sec = int(time.time())
    result_apply = end_epoch_sec - start_epoch_sec

    start_epoch_sec = int(time.time())
    df['result2'] = np.vectorize(divide)(df['A'], df['B'])
    end_epoch_sec = int(time.time())
    result_vectorize = end_epoch_sec - start_epoch_sec


    print 'N=%d, df.apply: %d sec, np.vectorize: %d sec' % \
            (N, result_apply, result_vectorize)

    # Make sure results from df.apply and np.vectorize match.
    assert(df['result'].equals(df['result2']))
```

The results are shown below:

```
N=1000, df.apply: 0 sec, np.vectorize: 0 sec

N=10000, df.apply: 1 sec, np.vectorize: 0 sec

N=100000, df.apply: 2 sec, np.vectorize: 0 sec

N=1000000, df.apply: 24 sec, np.vectorize: 1 sec

N=10000000, df.apply: 262 sec, np.vectorize: 4 sec
```

If `np.vectorize()` is in general always faster than `df.apply()`, then why is `np.vectorize()` not mentioned more? I only ever see StackOverflow posts related to `df.apply()`, such as:

pandas create new column based on values from other columns

How do I use Pandas 'apply' function to multiple columns?

How to apply a function to two columns of Pandas dataframe

python arrays pandas performance numpy

Share Edit Follow Flag

edited Oct 6, 2018 at 2:05

asked Oct 5, 2018 at 21:07

stackoverflowuser2010
33.8k 36 159 199

I didnt dig into the details of you question but `np.vectorize` is basically a python `for` loop (it's a convenience method) and `apply` with a lambda is also in python time – roganjosh Oct 5, 2018 at 21:30

"If np.vectorize() is in general always faster than df.apply(), then why is np.vectorize() not mentioned more?" Because you shouldn't be using `apply` on a row-by-row basis unless you have to, and obviously a vectorized function will out-perform a non-vectorized one. – PMende Oct 5, 2018 at 21:41

2 @PMende but `np.vectorize` is not vectorized. It's a well-known misnomer – roganjosh Oct 5, 2018 at 21:43

1 @PMende, Sure, I didn't imply otherwise. You shouldn't derive your opinions on implementation from timings. Yes, they're insightful. But they can make you presume things that aren't true. – jpp Oct 5, 2018 at 22:10

3 @PMende have a play with pandas `.str` accessors. They're slower than list comprehensions in a lot of cases. We assume too much. – roganjosh Oct 5, 2018 at 22:22

2 Answers

Sorted by: Highest score (default)

189

+50

I will *start* by saying that the power of Pandas and NumPy arrays is derived from high-performance **vectorised** calculations on numeric arrays.[1] The entire point of vectorised calculations is to avoid Python-level loops by moving calculations to highly optimised C code and utilising contiguous memory blocks.[2]

Python-level loops

Now we can look at some timings. Below are **all** Python-level loops which produce either `pd.Series`, `np.ndarray` or `list` objects containing the same values. For the purposes of assignment to a series within a dataframe, the results are comparable.

```
# Python 3.6.5, NumPy 1.14.3, Pandas 0.23.0

np.random.seed(0)
N = 10**5

%timeit list(map(divide, df['A'], df['B']))                                   # 43.9 ms
%timeit np.vectorize(divide)(df['A'], df['B'])                                # 48.1 ms
%timeit [divide(a, b) for a, b in zip(df['A'], df['B'])]                      # 49.4 ms
%timeit [divide(a, b) for a, b in df[['A', 'B']].itertuples(index=False)]     # 112 ms
%timeit df.apply(lambda row: divide(*row), axis=1, raw=True)                  # 760 ms
%timeit df.apply(lambda row: divide(row['A'], row['B']), axis=1)              # 4.83 s
%timeit [divide(row['A'], row['B']) for _, row in df[['A', 'B']].iterrows()]  # 11.6 s
```

Some takeaways:

1. The `tuple`-based methods (the first 4) are a factor more efficient than `pd.Series`-based methods (the last 3).
2. `np.vectorize`, list comprehension + `zip` and `map` methods, i.e. the top 3, all have roughly the same performance. This is because they use `tuple` *and* bypass some Pandas overhead from `pd.DataFrame.itertuples`.
3. There is a significant speed improvement from using `raw=True` with `pd.DataFrame.apply` versus without. This option feeds NumPy arrays to the custom function instead of `pd.Series` objects.

`pd.DataFrame.apply`: just another loop

To see *exactly* the objects Pandas passes around, you can amend your function trivially:

```
def foo(row):
```

```
def foo(row):
    print(type(row))
    assert False  # because you only need to see this once
df.apply(lambda row: foo(row), axis=1)
```

Output: <class 'pandas.core.series.Series'> . Creating, passing and querying a Pandas series object carries significant overheads relative to NumPy arrays. This shouldn't be surprise: Pandas series include a decent amount of scaffolding to hold an index, values, attributes, etc.

Do the same exercise again with `raw=True` and you'll see <class 'numpy.ndarray'> . All this is described in the docs, but seeing it is more convincing.

`np.vectorize` : fake vectorisation

The docs for [`np.vectorize`] has the following note:

> The vectorized function evaluates `pyfunc` over successive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

The "broadcasting rules" are irrelevant here, since the input arrays have the same dimensions. The parallel to `map` is instructive, since the `map` version above has almost identical performance. The source code shows what's happening: `np.vectorize` converts your input function into a Universal function ("ufunc") via `np.frompyfunc` . There is some optimisation, e.g. caching, which can lead to some performance improvement.

In short, `np.vectorize` does what a Python-level loop *should* do, but `pd.DataFrame.apply` adds a chunky overhead. There's no JIT-compilation which you see with `numba` (see below). It's just a convenience.

True vectorisation: what you *should* use

Why aren't the above differences mentioned anywhere? Because the performance of truly vectorised calculations make them irrelevant:

```
%timeit np.where(df['B'] == 0, 0, df['A'] / df['B'])       # 1.17 ms
%timeit (df['A'] / df['B']).replace([np.inf, -np.inf], 0)  # 1.96 ms
```

Yes, that's ~40x faster than the fastest of the above loopy solutions. Either of these are acceptable. In my opinion, the first is succinct, readable and efficient. Only look at other methods, e.g. `numba` below, if performance is critical and this is part of your bottleneck.

`numba.njit` : greater efficiency

numba.njit: greater efficiency

When loops *are* considered viable they are usually optimised via `numba` with underlying NumPy arrays to move as much as possible to C.

Indeed, `numba` improves performance to *microseconds*. Without some cumbersome work, it will be difficult to get much more efficient than this.

```
from numba import njit

@njit
def divide(a, b):
    res = np.empty(a.shape)
    for i in range(len(a)):
        if b[i] != 0:
            res[i] = a[i] / b[i]
        else:
            res[i] = 0
    return res

%timeit divide(df['A'].values, df['B'].values)  # 717 µs
```

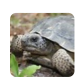
Using `@njit(parallel=True)` may provide a further boost for larger arrays.

[1] Numeric types include: `int`, `float`, `datetime`, `bool`, `category`. They *exclude* `object` dtype and can be held in contiguous memory blocks.

[2] There are at least 2 reasons why NumPy operations are efficient versus Python:

- Everything in Python is an object. This includes, unlike C, numbers. Python types therefore have an overhead which does not exist with native C types.
- NumPy methods are usually C-based. In addition, optimised algorithms are used where possible.

Share Edit Following Flag   edited Oct 8, 2018 at 22:19   answered Oct 5, 2018 at 23:38



jpp
147k 31 243 301

As a comment on "Creating, passing and querying a Pandas series object carries significant overheads relative to NumPy arrays." Compare: `%timeit [divide(a, b) for a, b in zip(df['A'], df['B'])]` results in: `16.4 ms ± 192 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)` versus: `%timeit [divide(a, b) for a, b in zip(df['A'].values, df['B'].values)]` results in `34.8 ms ± 388 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)`. This difference is consistent even if you pull the access of the `.values` attribute out of the loop. – PMende Oct 6, 2018 at 0:00

@PMende, You missed the point here, the series created with `apply` are **row-wise**, i.e. one element from `A` and one from `B` for *each* series. With the list comprehensions, `df['A']` / `df['B']` are the *only* 2 series and they aren't "created" in any sense, they already exist. `zip` can be compared to producing a `tuple`, which is much cheaper. – jpp Oct 6, 2018 at 0:01

12

The more complex your functions get (i.e., the less `numpy` can move to its own internals), the more you will see that the performance won't be that different. For example:

```
name_series = pd.Series(np.random.choice(['adam', 'chang', 'eliza', 'odom'],
replace=True, size=100000))

def parse_name(name):
    if name.lower().startswith('a'):
        return 'A'
    elif name.lower().startswith('e'):
        return 'E'
    elif name.lower().startswith('i'):
        return 'I'
    elif name.lower().startswith('o'):
        return 'O'
    elif name.lower().startswith('u'):
        return 'U'
    return name

parse_name_vec = np.vectorize(parse_name)
```

Doing some timings:

Using Apply

```
%timeit name_series.apply(parse_name)
```

Results:

```
76.2 ms ± 626 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Using `np.vectorize`

```
%timeit parse_name_vec(name_series)
```

Results:

```
77.3 ms ± 216 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Numpy tries to turn python functions into numpy `ufunc` objects when you call `np.vectorize`. How it does this, I don't actually know - you'd have to dig more into the internals of numpy than I'm willing to ATM. That said, it seems to do a better job on simply numerical functions than this string-based function here.

Cranking the size up to 1,000,000:

```
name_series = pd.Series(np.random.choice(['adam', 'chang', 'eliza', 'odom'],
replace=True, size=1000000))
```

`apply`

```
%timeit name_series.apply(parse_name)
```

Results:

```
769 ms ± 5.88 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

`np.vectorize`

```
%timeit parse_name_vec(name_series)
```

Results:

```
794 ms ± 4.85 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

A better (*vectorized*) way with `np.select` :

```
cases = [
    name_series.str.lower().str.startswith('a'),
name_series.str.lower().str.startswith('e'),
    name_series.str.lower().str.startswith('i'),
name_series.str.lower().str.startswith('o'),
    name_series.str.lower().str.startswith('u')
]
replacements = 'A E I O U'.split()
```

Timings:

```
%timeit np.select(cases, replacements, default=name_series)
```

Results:

```
67.2 ms ± 683 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Share Edit Follow Flag   edited Oct 5, 2018 at 22:47   answered Oct 5, 2018 at 22:32

PMende
4,551 2 15 24

---

2 I'm pretty sure your assertions here are incorrect. I can't back that statement up with code for