**Pandas vectorization: faster code, slower code, bloated memory**

by [Itamar Turner-Trauring](mailto:itamar@pythonspeed.com)  
Last updated 10 Jun 2022, originally created 09 Jun 2022

<https://pythonspeed.com/articles/pandas-vectorization/>?

When you’re processing data with Pandas, so-called “vectorized” operations can significantly speed up your code. Or at least, that’s the theory.

In practice, in some situations Pandas vectorized operations can actually make your code slower, or at least no faster. And they can also significantly increase memory usage.

Let’s dig in and see what vectorization means in Pandas, when and why it helps, and when it’s harmful.

*Vectorization: what it means, and how it speeds up your code*

Vectorization can mean different things, as discussed in a more in-depth article on [what vectorization means in Python](https://pythonspeed.com/articles/vectorization-python/). For our purposes there are two relevant meanings:

1. **Batch API**: An API that can process multiple items of data at once.
2. **A native-code loop:** In addition to exposing a batch API, the implementation runs quickly by not calling back into Python.

Importantly, in the context of Pandas’ documentation, vectorization only guarantees the first definition: the ability to run an operation across a whole Series, Index, or even DataFrame at once. In some cases, though, the APIs also implement the second defintion as well.

Consider the following semantically equivalent calculations:

# ... Vectorized operation:

df["ratio"] = 100 \* (df["x"] / df["y"])

# ... Non-vectorized operation:

def calc\_ratio(row):

return 100 \* (row["x"] / row["y"])

df["ratio2"] = df.apply(calc\_ratio, axis=1)

If we measure how long each takes to run, the result is:

Vectorized: 0.0043 secs

Non-vectorized: 5.6435 secs

**The vectorized 100 \* (df["x"] / df["y"]) is much faster because it avoids using Python code in the inner loop.** Internally, Pandas Series are often stored as NumPy arrays, in this case arrays of floats. Pandas is smart enough to pass the multiplication and division on to the underlying arrays, which then do a loop in machine code to do the multiplication. No slow Python code is involved in doing the arithmetic.

**In contrast, the non-vectorized method calls a Python function for every row, and that Python function does additional operations.** Eventually this devolves into low-level multiplication and division, but there is slow and expensive Python code being called repeatedly for every single row.

Our initial attempt at using vectorization appears to be a complete success: the code runs vastly faster.

*Vectorization in strings*

Let’s see how vectorization does on strings: Pandas provides a .str object on Series that lets you run various vectorized operations on strings. As an example, we’re going to calculate how many words there are in each sentence in a Series:

# ... Vectorized operation:

df["sentence\_length"] = df["sentences"].str.split().apply(

len

)

# ... Non-vectorized operation:

def sentence\_length(s):

return len(s.split())

df["sentence\_length2"] = df["sentences"].apply(

sentence\_length

)

The resulting run times:

Vectorized: 1.492 secs

Non-vectorized: 0.280 secs

The vectorized code is much slower! What’s going on?

Let’s measure the code with the [Sciagraph profiler](https://sciagraph.com/) and see what’s going on. Here’s a timeline of execution:

It looks like in the vectorized path the code [eventually devolves](https://github.com/pandas-dev/pandas/blob/v1.4.2/pandas/core/strings/object_array.py#L319) to a Python function that is applied to every row—just like our code:

def \_str\_split(

self,

pat: str | re.Pattern | None = None,

n=-1,

expand=False,

regex: bool | None = None,

):

# ...

f = lambda x: x.split(pat, n)

# ...

return self.\_str\_map(f, dtype=object)

So in this case, unlike the numeric calculations, the underlying implementation is in the end the same Python code, with a lot of overhead added on.

**This is why it’s so important to understand that vectorization has**[**multiple meanings**](https://pythonspeed.com/articles/vectorization-python/)**.** Vectorization in Pandas doesn’t necessarily mean the code will run faster, it just means the API lets you operate in batches, in this case on a whole Series. Sometimes that means it’ll also be faster, by running the whole loop with fast native code—the second meaning of vectorization—as was the case with the numeric calculations above. For strings, the code can be slower.

*The memory-use implications of vectorization*

So far we’ve been focusing on performance, but memory usage can also be a bottleneck. Use too much memory, and your program [might swap and start running slowly, or even crash](https://pythonspeed.com/articles/python-out-of-memory/).

How does vectorization impact memory use? Using the [Fil memory profiler](https://pythonspeed.com/fil/), I measured memory usage of each of the two methods we used to count the number of words in sentences. Here’s the vectorized code:

And here’s the non-vectorized code:

The vectorized code uses vastly more memory. Inspecting the profiling result above, and reading the original code, we can see what’s going on: a temporary Series is being created, containing lists of words. The code we wrote is equivalent to:

temporary\_series = df["sentences"].str.split()

df["sentence\_length"] = temporary\_series.apply(len)

This temporary Series is massively increasing memory usage, by storing [heavyweight Python objects](https://pythonspeed.com/articles/python-object-memory/) (lists and strings), and we don’t even need it. In the non-vectorized code we split a sentence string into a list, run len() on that, and then throw away the list. So we only use temporary memory for the words of a single sentence at time, rather than all sentences at once: O(1) instead of O(N). This results in far less memory usage.

*Takeaways*

1. Be aware of the [multiple meanings of vectorization](https://pythonspeed.com/articles/vectorization-python/). In Pandas, it just means a batch API.
2. Numeric code in Pandas often benefits from the second meaning of vectorization, a vastly faster native code loop.
3. Vectorization in strings in Pandas can often be slower, since it doesn’t use native code loops.
4. Vectorization can result in temporary Series, with a corresponding increase in memory usage proportional to the Series size.

More broadly: if you care about performance and memory usage, you need to measure it! Some results above I did not expect; some of the solutions I tried didn’t work and were therefore omitted.

**Without measurement, you have no insight in to where the bottlenecks are.** For offline memory profiling, you can use the open source [Fil profiler](https://pythonspeed.com/fil/) and for offline performance profiling you can use [py-spy](https://github.com/benfred/py-spy) or [Austin](https://github.com/P403n1x87/austin). For production performance and memory profiling of Python batch jobs, try out [Sciagraph](https://sciagraph.com/).

**Learn practical Python software engineering skills you can use at your job**

Too much to learn? Don't know where to start?

Sign up for my newsletter, and join over 5900 Python developers and data scientists learning practical tools and techniques, from Docker packaging to testing to Python best practices, with a free new article in your inbox every week.

Top of Form



Level up your Python engineering skills

Bottom of Form

**Next:** [Where’s your bottleneck? CPU time vs wallclock time](https://pythonspeed.com/articles/blocking-cpu-or-io/)  
**Previous:** [CPUs, cloud VMs, and noisy neighbors: the limits of parallelism](https://pythonspeed.com/articles/cpu-limits-to-speed/)