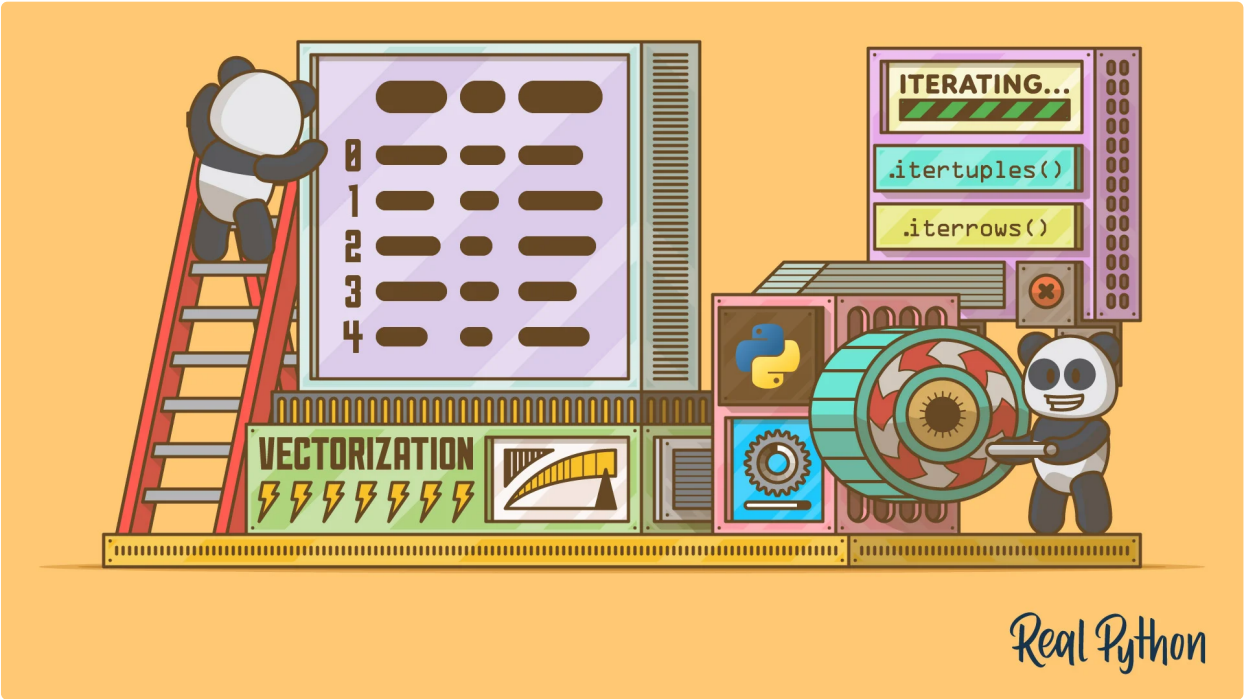


# How to Iterate Over Rows in pandas, and Why You Shouldn't

by Ian Currie Jan 30, 2023 3 Comments best-practices data-science intermediate

Mark as Completed

Tweet Share Email

## Table of Contents


- How to Iterate Over DataFrame Rows in pandas
- Why You Should Generally Avoid Iterating Over Rows in pandas

- Using Vectorized Methods Over Iteration
- Use Intermediate Columns So You Can Use Vectorized Methods
- Conclusion




One of the most common questions you might have when entering the world of pandas is *how to iterate over rows* in a pandas DataFrame. If you've gotten comfortable using loops in core Python, then this is a perfectly natural question to ask.

While iterating over rows is relatively straightforward with `.itertuples()` or `.iterrows()`, that doesn't necessarily mean iteration is the best way to work with DataFrames. In fact, while iteration may be a quick way to make progress, relying on iteration can become a significant roadblock when it comes to being effective with pandas.

In this tutorial, you'll learn how to iterate over the rows in a pandas DataFrame, but you'll also learn why you probably don't want to. Generally, you'll want to avoid iteration because it comes with a performance penalty and goes against the way of the panda.

To follow along with this tutorial, you can download the datasets and code samples from the following link:

**Free Sample Code: Click here to download the free sample code and datasets** that you'll use to explore iterating over rows in a pandas DataFrame vs using vectorized methods.

The last bit of prep work is to spin up a virtual environment and install a few packages:

Windows | Linux + macOS



Table of Contents


- How to Iterate Over DataFrame Rows in pandas
- Why You Should Generally Avoid Iterating Over Rows in pandas
- Using Vectorized Methods Over Iteration
- Use Intermediate Columns So You Can Use Vectorized Methods
- Conclusion

Windows PowerShell

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install pandas httpx codetiming
```

The `pandas` installation won't come as a surprise, but you may wonder about the others. You'll use the `httpx` package to carry out some HTTP requests as part of one example, and the `codetiming` package to make some quick performance comparisons.

With that, you're ready to get stuck in and learn how to iterate over rows, why you probably don't want to, and what other options to rule out before resorting to iteration.

## How to Iterate Over DataFrame Rows in pandas

While uncommon, there are some situations in which you can get away with iterating over a DataFrame. These situations are typically ones where you:

- Need to feed the information from a pandas DataFrame sequentially into another **API**
- Need the operation on each row to produce a **side effect**, such as an HTTP request
- Have **complex operations** to carry out involving various columns in the DataFrame
- Don't mind the **performance penalty** of iteration, maybe because working with the data isn't the bottleneck, the dataset is very small, or it's just a personal project

A common use case for using loops in pandas is when you are **interactively** exploring and experimenting with data. In these cases, performance is usually less of a concern. By iterating over the data rows, you can display and get to know individual rows. Based on this experience you can implement more effective approaches later.

As an example of a more permanent use case, imagine you have a list of URLs in a

DataFrame, and you want to check which URLs are online. In the downloadable materials,

you'll find a CSV file with some data on the most popular websites, which you can load into a DataFrame:

Python >>>

```
>>> import pandas as pd
>>> websites = pd.read_csv("resources/popular_websites.csv", index_col=0)
>>> websites
         name                              url   total_views
0      Google           https://www.google.com  5.207268e+11
1     YouTube          https://www.youtube.com  2.358132e+11
2    Facebook         https://www.facebook.com  2.230157e+11
3       Yahoo            https://www.yahoo.com  1.256544e+11
4   Wikipedia        https://www.wikipedia.org  4.467364e+10
5       Baidu            https://www.baidu.com  4.409759e+10
6     Twitter              https://twitter.com  3.098676e+10
7      Yandex               https://yandex.com  2.857980e+10
8   Instagram        https://www.instagram.com  2.621520e+10
9         AOL              https://www.aol.com  2.321232e+10
10   Netscape         https://www.netscape.com  5.750000e+06
11       Nope  https://alwaysfails.example.com  0.000000e+00
```

This data contains the website's name, its URL, and the total number of views over an unspecified time period. In the example, pandas shows the number of views in scientific notation. You've also got a dummy website in there for testing purposes.

You want to write a connectivity checker to test the URLs and provide a human-readable message indicating whether the website is online or whether it's being redirected to another URL:

Python >>>

```
>>> import httpx
>>> def check_connection(name, url):
...     try:
...         response = httpx.get(url)
...         location = response.headers.get("location")
...         if location is None or location.startswith(url):
...             print(f"{name} is online!")
...         else:
...             print(f"{name} is online! But redirects to {location}")
...         return True
...     except httpx.ConnectError:
...         print(f"Failed to establish a connection with {url}")
...         return False
...
```

Here, you've defined a `check_connection()` function to make the request and print out messages for a given name and URL.

With this function, you'll use both the `url` and the `name` columns. You don't care much about the performance of reading the values from the DataFrame for two reasons—partly because the data is so small, but mainly because the real time sink is making HTTP requests, not reading from a DataFrame.

Additionally, you're interested in inspecting whether any of the websites are down. That is, you're interested in the *side effect* and not in *adding information* to the DataFrame.

For these reasons, you can get away with using `.itertuples()`:

Python >>>

```
>>> for website in websites.itertuples():
...     check_connection(website.name, web.url)
...
Google is online!
YouTube is online!
Facebook is online!
Yahoo is online!
Wikipedia is online!
Baidu is online!
Twitter is online!
Yandex is online!
Instagram is online!
AOL is online!
Netscape is online! But redirects to https://www.aol.com/
Failed to establish a connection with https://alwaysfails.example.com
```

Here you use a `for` loop on the iterator that you get from `.itertuples()`. The iterator yields a namedtuple for each row. Using dot notation, you select the two columns to feed into the `check_connection()` function.

**Note:** If, for any reason, you want to use dynamic values to select columns from each row, then you can use `.iterrows()`, even though it's slightly slower. The `.iterrows()` method returns a two-item tuple of the index number and a `Series` object for each row. The same iteration as above would look like this with `.iterrows()`:

Python

```
for _, website in websites.iterrows():
    check_connection(website["name"], website["url"])
```

In this code, you discard the index number from each tuple produced by `.iterrows()`.

Then with the `Series` object, you can use square bracket (`[]`) indexing to select the

columns that you need from each row. Square bracket indexing allows you to use any expression, such as a variable, within the square brackets.

In this section, you've looked at how to iterate over a pandas DataFrame's rows. While iteration makes sense for the use case demonstrated here, you want to be careful about applying this knowledge elsewhere. It may be tempting to use iteration to accomplish many other types of tasks in pandas, but it's not the pandas way. Coming up, you'll learn the main reason why.



## Why You Should Generally Avoid Iterating Over Rows in pandas

The pandas library leverages array programming, or **vectorization**, to dramatically increase its performance. Vectorization is about finding ways to apply an operation to a set of values at once instead of one by one.

For example, if you had two lists of numbers and you wanted to add each item to the other, then you might create a `for` loop to go through and add each item to its counterpart:

Python >>>

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> for a_int, b_int in zip(a, b):
...     print(a_int + b_int)
...
5
7
9
```

While looping is a perfectly valid approach, pandas and some of the libraries it depends on—like NumPy—leverage array programming to be able to operate on the whole list in a much more efficient way.

Vectorized functions make it seem like you're operating on the entire list in one operation. With this way of thinking, it allows the libraries to leverage concurrency, special processor and memory hardware, and low-level compiled languages like C.

All of these techniques and more make vectorized operations significantly faster than explicit loops when one operation has to be applied to a sequence of items. For example, pandas encourages you to look at operations as things that you apply to *entire columns at once*, not one row at a time.

Using vectorized operations on tabular data is what makes pandas, pandas. You should always seek out vectorized operations first. There are many `DataFrame` and `Series` methods to choose from, so keep the superb pandas documentation handy.

Since vectorization is an integral part of pandas, you'll often hear people say *if you're looping in pandas, then you're doing it wrong*. Or perhaps even something more extreme, from a wonderful article by @ryxcommar:

> Loops in pandas are a sin. (Source)

While these pronouncements may be exaggerated for effect, they're a good rule of thumb if you're new to pandas. Almost everything that you need to do with your data is possible with vectorized methods. If there's a specific method for your operation, then it's usually best to use that method—for speed, for reliability, and for readability.

Similarly, in the fantastic StackOverflow pandas Canonicals put together by Coldsp33d, you'll find another measured warning against iteration:

> Iteration in Pandas is an anti-pattern and is something you should only do when you have exhausted every other option. (Source)

Check out the canonicals for more performance metrics and information about what other options are available.

Basically, when you're using pandas for what it's designed for—data analysis and other data-wrangling operations—you can almost always rely on vectorized operations. But sometimes you need to code on the outskirts of pandas territory, and that's when you might get away with iteration. This is the case when interfacing with other APIs, for instance, to make HTTP requests, as you did in the earlier example.

Adopting the vectorized mindset may seem a bit strange to begin with. Much of learning about programming involves learning about iteration, and now you're being told that you need to think of an operation happening on a sequence of items *at the same time*? What kind of sorcery is this? But if you're going to be using pandas, then embrace vectorization, and be rewarded with high-performance, clean, and idiomatic pandas.

In the next section, you'll walk through a couple of examples that pit iteration against vectorization, and you'll compare their performance.

# Using Vectorized Methods Over Iteration

In this section and the next, you'll be looking at examples of when you might be tempted to use an iterative approach, but where vectorized methods are significantly faster.

Say you wanted to take the sum of all the views in the website dataset that you were working with earlier in this tutorial.

To take an iterative approach, you could use `.itertuples()`:

Python >>>

```
>>> import pandas as pd
>>> websites = pd.read_csv("resources/popular_websites.csv", index_col=0)
>>> total = 0
>>> for website in websites.itertuples():
...     total += website.total_views
...
>>> total
1302975468008.0
```

This would represent an iterative approach to calculating a sum. You have a `for` loop that goes row by row, taking the value and incrementing a `total` variable. Now, you might recognize a more Pythonic approach to taking the sum:

Python >>>

```
>>> sum(website.total_views for website in websites.itertuples())
1302975468008.0
```

Here, you use the `sum()` built-in method along with a generator expression to take the sum.

While these may seem like decent approaches—and they certainly work—they're not idiomatic pandas, especially when you have the `.sum()` vectorized method available:

Python >>>

```
>>> websites["total_views"].sum()
1302975468008.0
```

Here you select the `total_views` column with square bracket indexing on the DataFrame. This indexing returns a `Series` object representing the `total_views` column. Then you use the `.sum()` method on the Series.

The most evident advantage of this method is that it's arguably the most readable of the three. But its readability, while immensely important, isn't the most dramatic advantage.

Inspect the script below, where you're using the `codetiming` package to compare the three methods:

Python

```python
# take_sum_codetiming.py

import pandas as pd
from codetiming import Timer

def loop_sum(websites):
    total = 0
    for website in websites.itertuples():
        total += website.total_views
    return total

def python_sum(websites):
    return sum(website.total_views for website in websites.itertuples())

def pandas_sum(websites):
    return websites["total_views"].sum()

for func in [loop_sum, python_sum, pandas_sum]:
    websites = pd.read_csv("resources/popular_websites.csv", index_col=0)
    with Timer(name=func.__name__, text="{name:20}: {milliseconds:.2f} ms"):
        func(websites)
```

In this script, you define three functions, all of which take the sum of the `total_views` column. All the functions accept a DataFrame and return a sum, but they use the following three approaches, respectively:

1. A `for` loop and `.itertuples()`
2. The Python `sum()` function and a comprehension using `.itertuples()`
3. The pandas `.sum()` vectorized method

These are the three approaches that you explored above, but now you're using `codetiming.Timer` to learn how quickly each function runs.

`codetiming.Timer` to learn how quickly each function runs.

Your precise results will vary, but the proportion should be similar to what you can see below:

Shell

```
$ python take_sum_codetiming.py
loop_sum             : 0.24 ms
python_sum           : 0.19 ms
pandas_sum           : 0.14 ms
```

Even for a tiny dataset like this, the difference in performance is quite drastic, with pandas' `.sum()` being nearly twice as fast as the loop. Python's built-in `sum()` is an improvement over the loop, but it's still no match for pandas.

> **Note:** `codetiming` is designed to make it convenient to monitor the runtime of your production code. When using the library for benchmarking, like you're doing here, you should run your code a few times to check the stability of your timings.

That said, with a dataset this tiny, it doesn't quite do justice to the scale of optimization that vectorization can achieve. To take things to the next level, you can artificially inflate the dataset by duplicating the rows one thousand times, for example:

File Changes (diff)

```
 # python take_sum_codetiming.py

 # ...

 for func in [pandas_sum, loop_sum, python_sum]:
     websites = pd.read_csv("resources/popular_websites.csv", index_col=0)
+    websites = pd.concat([websites for _ in range(1000)])
     with Timer(name=func.__name__, text="{name:20}: {milliseconds:.2f} ms"):
```

```
        func(websites)
```

This modification uses the `concat()` function to concatenate one thousand instances of `websites` with each other. Now you've got a dataset of a few thousand rows. Running the timing script again will yield results similar to the these:

Shell

```
$ python take_sum_codetiming.py
loop_sum            : 3.55 ms
python_sum          : 3.67 ms
pandas_sum          : 0.15 ms
```

It seems that the pandas `.sum()` method still takes around the same amount of time, while the loop and Python's `sum()` have increased a great deal more. Note that pandas' `.sum()` is around twenty times faster than plain Python loops!

In the next section, you'll see an example of how to work in a vectorized manner, even if pandas doesn't offer a specific vectorized method for your task.



## Use Intermediate Columns So You Can Use Vectorized Methods

You might hear that it's okay to use iteration when you have to use multiple columns to get the result that you need. Take, for instance, a dataset that represents sales of product per month:

Python >>>

```
>>> import pandas as pd
>>> products = pd.read_csv("resources/products.csv")
>>> products
      month  sales  unit_price
0   january      3        0.50
1  february      2        0.53
2     march      5        0.55
3     april     10        0.71
4       may      8        0.66
```

This data shows columns for the number of sales and the average unit price for a given month. But what you need is the **cumulative sum** of the total income for several months.

You may know that pandas has a `.cumsum()` method to take the cumulative sum. But in this case, you'll have to multiply the `sales` column by the `unit_price` first to get the total sales for each month.

This situation may tempt you down the path of iteration, but there's a way to get around these limitations. You can use **intermediate columns**, even if it means running two vectorized operations. In this case, you'd multiply `sales` and `unit_price` first to get a new column, and then use `.cumsum()` on the new column.

Consider this script, where you're comparing the performance of these two approaches by generating a DataFrame with an extra `cumulative_sum` column:

Python

```python
# cumulative_sum_codetiming.py

import pandas as pd
from codetiming import Timer

def loop_cumsum(products):
    cumulative_sum = []
    for product in products.itertuples():
        income = product.sales * product.unit_price
        if cumulative_sum:
            cumulative_sum.append(cumulative_sum[-1] + income)
        else:
            cumulative_sum.append(income)
    return products.assign(cumulative_income=cumulative_sum)

def pandas_cumsum(products):
    return products.assign(
        income=lambda df: df["sales"] * df["unit_price"],
        cumulative_income=lambda df: df["income"].cumsum(),
    ).drop(columns="income")

for func in [loop_cumsum, pandas_cumsum]:
    products = pd.read_csv("resources/products.csv")
    with Timer(name=func.__name__, text="{name:20}: {milliseconds:.2f} ms"):
        func(products)
```

In this script, you aim to add a column to the DataFrame, and so each function accepts a DataFrame of `products` and will use the `.assign()` method to return a DataFrame with a new column called `cumulative_sum`.

The `.assign()` method takes keyword arguments, which will be the names of columns. They can be names that don't yet exist in the DataFrame, or ones that already exist. If the

columns already exist, then pandas will update them.

The value of each keyword argument can be a callback function that takes a DataFrame and returns a Series. In the example above, in the `pandas_cumsum()` function, you use lambda functions as callbacks. Each callback returns a new Series.

In `pandas_cumsum()`, the first callback creates the `income` column by multiplying the columns of `sales` and `unit_price` together. The second callback calls `.cumsum()` on the new `income` column. After these operations are done, you use the `.drop()` method to discard the intermediate `income` column.

Running this script will produce results similar to these:

Shell

```
$ python cumulative_sum_codetiming.py
loop_cumsum         : 0.43 ms
pandas_cumsum       : 1.04 ms
```

Wait, the loop is actually faster? Wasn't the vectorized method meant to be faster?

As it turns out, for absolutely tiny datasets like these, the overhead of doing two vectorized operations—multiplying two columns, then using the `.cumsum()` method—is slower than iterating. But, go ahead and bump up the numbers in the same way you did for the previous test:

File Changes (diff)

```
 for f in [loop_cumsum, pandas_cumsum]:
     products = pd.read_csv("resources/products.csv")
+    products = pd.concat(products for _ in range(1000))
     with Timer(name=f.__name__, text="{name:20}: {milliseconds:.2f} ms"):
```

Running with a dataset one thousand times larger will reveal much the same story as with

`.sum()`:

Shell

```
$ python cumulative_sum_codetiming.py
loop_cumsum         : 2.80 ms
pandas_cumsum       : 1.21 ms
```

pandas pulls ahead again, and will keep pulling ahead more dramatically as your dataset gets larger. Even though it has to do two vectorized operations, once your dataset gets larger than a few hundred rows, pandas leaves iteration in the dust.

Not only that, but you end up with beautiful, idiomatic pandas code, which other pandas professionals will recognize and be able to read quickly. While it may take a little while to get used this way of writing code, you'll never want to go back!



## Conclusion

In this tutorial, you've learned how to iterate over the rows of a DataFrame and when such an approach might make sense. But you've also learned about why you probably don't want to do this most of the time. You've learned about vectorization and how to look for ways to used vectorized methods instead of iterating—and you've ended up with beautiful, blazing-fast, idiomatic pandas.

**Free Sample Code: Click here to download the free sample code and datasets** that you'll use to explore iterating over rows in a pandas DataFrame vs using

vectorized methods.

Check out the downloadable materials, where you'll find another example comparing the performance of vectorized methods with other alternatives, including some list comprehensions that actually beat a vectorized operation.

Mark as Completed

## 🐍 Python Tricks 💌

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

## About **Ian Currie**

Ian is a Python nerd who uses it for everything from tinkering to helping people and companies manage their day-to-day and

develop their businesses.

» More about Ian

*Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:*

Aldren Geir Arne Kate

## What Do You Think?

**Rate this article:**

Tweet Share Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

**Commenting Tips:** The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

## Keep Learning

Related Tutorial Categories: best-practices data-science intermediate

❤️ Happy Pythoning!