towards
data science

Follow 592K Followers

# 6 Pandas Mistakes that Silently Tell You Are a Rookie

No error messages — that's what makes them subtle

Bex T. 1 day ago · 7 min read★


Photo by Michal Matlon on Unsplash. All images are by the author unless specified otherwise.

## Introduction

We are all used to the big, fat, red error messages that frequently pop up while we code. Fortunately, people won't usually see them because we always fix them. But how about the mistakes that give no errors? These are the most dangerous ones and would embarrass us the most when spotted by more experienced folks.

These mistakes are not related to the API or syntax of the tool you are using but are directly associated with theory and your experience level. Today, we are here to talk

about 6 of such mistakes that come up often among beginner Pandas users, and we will learn how to solve them.

. . .

## 1. Using Pandas itself

It is kind of ironic that the first mistake is related to actually using Pandas for certain tasks. Specifically, today's real-world tabular datasets are just massive. To read them into your environment with Pandas would be a huge mistake.

Why? Because it is so damn slow! Below, we load the TPS October dataset with 1M rows and ~300 features, taking up a whopping 2.2GB of disk space.

It took ~22 seconds. Now, you might be saying that 22 seconds isn't that much but imagine this. In a single project, you will perform many experiments during different stages. You will probably create separate scripts or notebooks for cleaning, feature engineering, choosing a model, and many more for other tasks.

Waiting for the data to load for 20 seconds multiple times really gets on your nerves. Besides, your dataset will probably be much larger. So, what is a faster solution?

The solution is to ditch Pandas at this stage and use other alternatives that are designed explicitly for fast IO. My favorite one is `datatable` but you can also go for `Dask`, `Vaex`, `cuDF`, etc. Here is how long it takes to load the same dataset with `datatable`:

Just 2 seconds!

. . .

## 2. No vectors?

One of the craziest rules in functional programming is never to use loops (along with the "no variables" rule). It seems that sticking to this "no-loops" rule while using Pandas is the best you can do to speed up computations.

Functional programming replaces loops with recursion. Fortunately, we don't have to be so hard on ourselves because we can just use vectorization!

Vectorization, which is at the heart of Pandas and NumPy, performs mathematical operations on whole arrays rather than individual scalars. The best part is that Pandas already has an extensive suite of vectorized functions, eliminating the need to reinvent the wheel.

All arithmetic operators in Python (+, -, *, /, **) work in vectorized manner when used on Pandas series or dataframes. Also, any other mathematical function you see in Pandas or NumPy is already vectorized.

To see the speed increase, we will use the below `big_function` that takes three columns as an input and performs some meaningless arithmetic:

First, we will use this function with Pandas's fastest iterator — `apply`:

The operation took 20 seconds. Let’s do the same by using the core NumPy arrays in a vectorized manner:

It took only 82 milliseconds, which is about 250 times faster.

Indeed, you can’t completely ditch loops. After all, not all data manipulation operations are mathematical. But whenever you find yourself itching to use some looping functions like `apply`, `applymap` or `itertuples`, take a moment to see if what you want to do can be vectorized.

. . .

## 3. Data types, dtypes, types!

No, this is not that “Change the default data types of Pandas columns” lesson you received in middle school. Here, we will go much deeper. Specifically, we discuss data types in terms of their memory usage.

The worst and most memory-consuming data type is `object`, which also happens to limit some of the features of Pandas. Next, we have floats and integers. Actually, I don't want to list all Pandas data types, so why don't you take a look at this table:

| Pandas dtype | Python type | NumPy type | Usage |
|---|---|---|---|
| object | str | string_, unicode_ | Text |
| int64 | int | int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64 | Integer numbers |
| float64 | float | float_, float16, float32, float64 | Floating point numbers |
| bool | bool | bool_ | True/False values |
| datetime64 | NA | datetime64[ns] | Date and time values |
| timedelta[ns] | NA | NA | Differences between two datetimes |
| category | NA | NA | Finite list of text values |

Source: http://pbpython.com/pandas_dtypes.html

After the data type name, the numbers represent how many bits of memory each number in this data type will take. So, the idea is to cast every column in our dataset to the smallest subtype as possible. How do you know which one to choose? Well, here is another table for you:



Source: https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html

Generally, you want to cast floats to `float16/32` and the columns with both positive and negative integers to `int8/16/32` based on the above table. You can also use `uint8` for booleans and positive-only integers for more decrease in memory consumption.

Here is a handy but long function that casts floats and integers to their smallest subtype based on the above table:

Let’s use it on the TPS October data and see how much reduction we can get:

| | id | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | ... | f278 | f279 | f280 | f281 | f282 | f283 | f284 | target | f1001 | f1000 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0.20593 | 0.41089 | 0.17676 | 0.22363 | 0.42358 | 0.47607 | 0.41357 | 0.61182 | 0.53467 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -2.59375 | -2.59375 |
| **1** | 1 | 0.18103 | 0.47314 | 0.01173 | 0.21362 | 0.61963 | 0.44165 | 0.23035 | 0.68604 | 0.28198 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -6.64453 | -6.64453 |
| **2** | 2 | 0.18262 | 0.30737 | 0.32593 | 0.20715 | 0.60547 | 0.30981 | 0.49341 | 0.75098 | 0.53613 | ... | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | -1.67285 | -1.67285 |
| **3** | 3 | 0.18030 | 0.49463 | 0.00837 | 0.22363 | 0.76074 | 0.43921 | 0.43213 | 0.77637 | 0.48389 | ... | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | -7.14844 | -7.14844 |
| **4** | 4 | 0.17712 | 0.49561 | 0.01426 | 0.54883 | 0.62549 | 0.56250 | 0.11719 | 0.56104 | 0.07709 | ... | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | -6.36328 | -6.36328 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **999995** | 999995 | 0.20435 | 0.34473 | 0.26221 | 0.22839 | 0.61084 | 0.35742 | 0.49048 | 0.61377 | 0.50928 | ... | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | -1.99414 | -1.99414 |
| **999996** | 999996 | 0.18201 | 0.56396 | 0.24255 | 0.24121 | 0.45361 | 0.46948 | 0.47754 | 0.65918 | 0.51904 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | -2.12500 | -2.12500 |
| **999997** | 999997 | 0.25024 | 0.49146 | 0.09857 | 0.23560 | 0.77148 | 0.36792 | 0.53174 | 0.59814 | 0.61865 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -3.45703 | -3.45703 |
| **999998** | 999998 | 0.20361 | 0.53516 | 0.18018 | 0.21313 | 0.65479 | 0.53516 | 0.31616 | 0.65234 | 0.39795 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -2.57031 | -2.57031 |
| **999999** | 999999 | 0.16101 | 0.59619 | 0.01306 | 0.28027 | 0.58008 | 0.40186 | 0.49390 | 0.61182 | 0.53125 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -6.50781 | -6.50781 |

1000000 rows × 289 columns

We compressed the dataset to 510 MBs from the original 2.2GB. Unfortunately, this decrease in memory consumption gets lost when we save the dataframe to file.

Why was this a mistake again? Well, RAM consumption plays a big part when working with such datasets using big machine learning models. Once you get a few OutOfMemory errors, you start to catch up and learn tricks like this to keep your computer happy.

. . .

## 4. No styling?

One of the most wonderful features of Pandas is its ability to display stylized dataframes. Raw dataframes are rendered as HTML tables with a bit of CSS inside Jupyter.

For people with style and who want to go the extra mile to make their notebooks more colorful and appealing, Pandas allows styling its DataFrames through the `style` attribute.

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| f227 | 1000000.000000 | 0.042318 | 0.097016 | 0.000041 | 0.006148 | 0.008488 | 0.011224 | 0.995261 |
| f143 | 1000000.000000 | 0.499120 | 0.218694 | 0.000000 | 0.506243 | 0.569084 | 0.622750 | 0.999710 |
| f234 | 1000000.000000 | 0.111832 | 0.106602 | 0.024656 | 0.035676 | 0.081907 | 0.140333 | 0.996664 |
| f166 | 1000000.000000 | 0.030398 | 0.094173 | 0.000000 | 0.006335 | 0.008709 | 0.011248 | 1.000000 |
| f90 | 1000000.000000 | 0.642243 | 0.141101 | 0.000000 | 0.539810 | 0.619562 | 0.778886 | 0.981087 |
| f84 | 1000000.000000 | 0.066512 | 0.104063 | 0.000000 | 0.012872 | 0.018155 | 0.097201 | 1.000000 |
| f121 | 1000000.000000 | 0.186078 | 0.040156 | 0.010368 | 0.169747 | 0.170958 | 0.191037 | 0.973322 |
| f243 | 1000000.000000 | 0.213172 | 0.409548 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 1.000000 |
| f30 | 1000000.000000 | 0.118508 | 0.099597 | 0.000000 | 0.056052 | 0.059557 | 0.194628 | 0.967574 |
| f67 | 1000000.000000 | 0.223665 | 0.077721 | 0.053498 | 0.174431 | 0.208975 | 0.247528 | 0.939481 |
| f258 | 1000000.000000 | 0.539352 | 0.498449 | 0.000000 | 0.000000 | 1.000000 | 1.000000 | 1.000000 |
| f168 | 1000000.000000 | 0.018734 | 0.066985 | 0.000000 | 0.005187 | 0.006973 | 0.008854 | 0.885041 |
| f51 | 1000000.000000 | 0.192615 | 0.049062 | 0.137986 | 0.178439 | 0.179516 | 0.180729 | 0.775959 |
| f66 | 1000000.000000 | 0.256157 | 0.075838 | 0.003643 | 0.189771 | 0.238813 | 0.287281 | 0.977055 |
| f68 | 1000000.000000 | 0.133033 | 0.123449 | 0.000137 | 0.012811 | 0.205306 | 0.211725 | 0.979451 |
| f96 | 1000000.000000 | 0.211714 | 0.141129 | 0.051701 | 0.159743 | 0.161914 | 0.164423 | 1.000000 |
| f44 | 1000000.000000 | 0.661738 | 0.271317 | 0.000667 | 0.432623 | 0.828384 | 0.835743 | 1.000000 |
| f157 | 1000000.000000 | 0.368812 | 0.290552 | 0.000505 | 0.036547 | 0.548539 | 0.618072 | 0.963641 |
| f123 | 1000000.000000 | 0.132864 | 0.103196 | 0.000000 | 0.032964 | 0.112700 | 0.196135 | 1.000000 |
| f240 | 1000000.000000 | 0.187746 | 0.061176 | 0.011777 | 0.142875 | 0.170167 | 0.199849 | 0.937160 |

Above, we randomly choose 20 columns, create a 5-number summary for them, transpose the result, and color the mean, std, and median columns based on their magnitude.

Changes like these make it easier to spot patterns in raw numbers without turning to visualization libraries. You can learn the full details of how you can style your dataframes from this link.

Actually, there is nothing wrong with *not styling* your dataframes. However, this seemed such a good feature that not using it would be a missed opportunity.

. . .

## 5. Saving to CSVs

Just like reading CSV files is extremely slow, so is saving the data back to them. Here is how long it takes to save the TPS October data to CSV:

It took almost 3 minutes. To be fair to everyone else and yourself, save your dataframes to some other lighter and cheaper format like feather or parquet.

As you can see, saving the dataframe to feather format took 160 times less runtime. Besides, feather and parquet also take much less storage. My favorite writer here, Dario Radečić has an entire series dedicated to CSV alternatives. You can check it out here.

. . .

## 6. You should've read the user guide!

Actually, the most grievous mistake in this list is not reading the User Guide or the documentation of Pandas.

I understand. We all have this weird thing when it comes to documentation. We'd rather scour the Internet for hours than read the docs.

However, this isn't at all true when it comes to Pandas. It has an excellent user guide

covering topics right from the basics to contributing and making Pandas more awesome.

In fact, you could've learned about all the mistakes I mentioned today from the user guide. It is even true that the section on reading large datasets specifically tells you to use other packages like `Dask` to read massive files and stay away from Pandas. If I had the time to read the user guide from start to finish, I would've probably come up with 50 more beginner mistakes, but now that you know what to do, I leave the rest to you.

. . .

## Summary

Today, we have learned about the six most common mistakes beginners make while using Pandas.

I want you to note that most of these mistakes are actually counted wrong when working with gigabyte-sized datasets. You might as well forget about them if you are still playing around with toy datasets because the solutions won't make much of a difference.

However, as you improve your skills and start tackling real-world datasets, the concepts will eventually be beneficial.



. . .

## Before you leave, my readers love these — why don't you give them a check?





Edit description

towardsdatascience.com

**7 Cool Python Packages Kagglers Are Using Without Telling You**

Edit description

towardsdatascience.com

**Why Is Everyone at Kaggle Obsessed with Optuna For Hyperparameter Tuning?**

Edit description

towardsdatascience.com

**How to Work with Million-row Datasets Like a Pro**

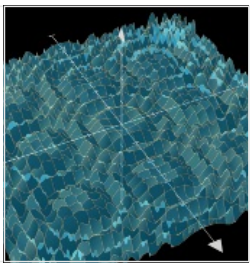
Edit description

towardsdatascience.com

**Love 3Blue1Brown Animations? Learn How to Create Your Own in Python in 10 Minutes**

Edit description

towardsdatascience.com



Data Science | Pandas | Towards Data Science | Programming | Machine Learning