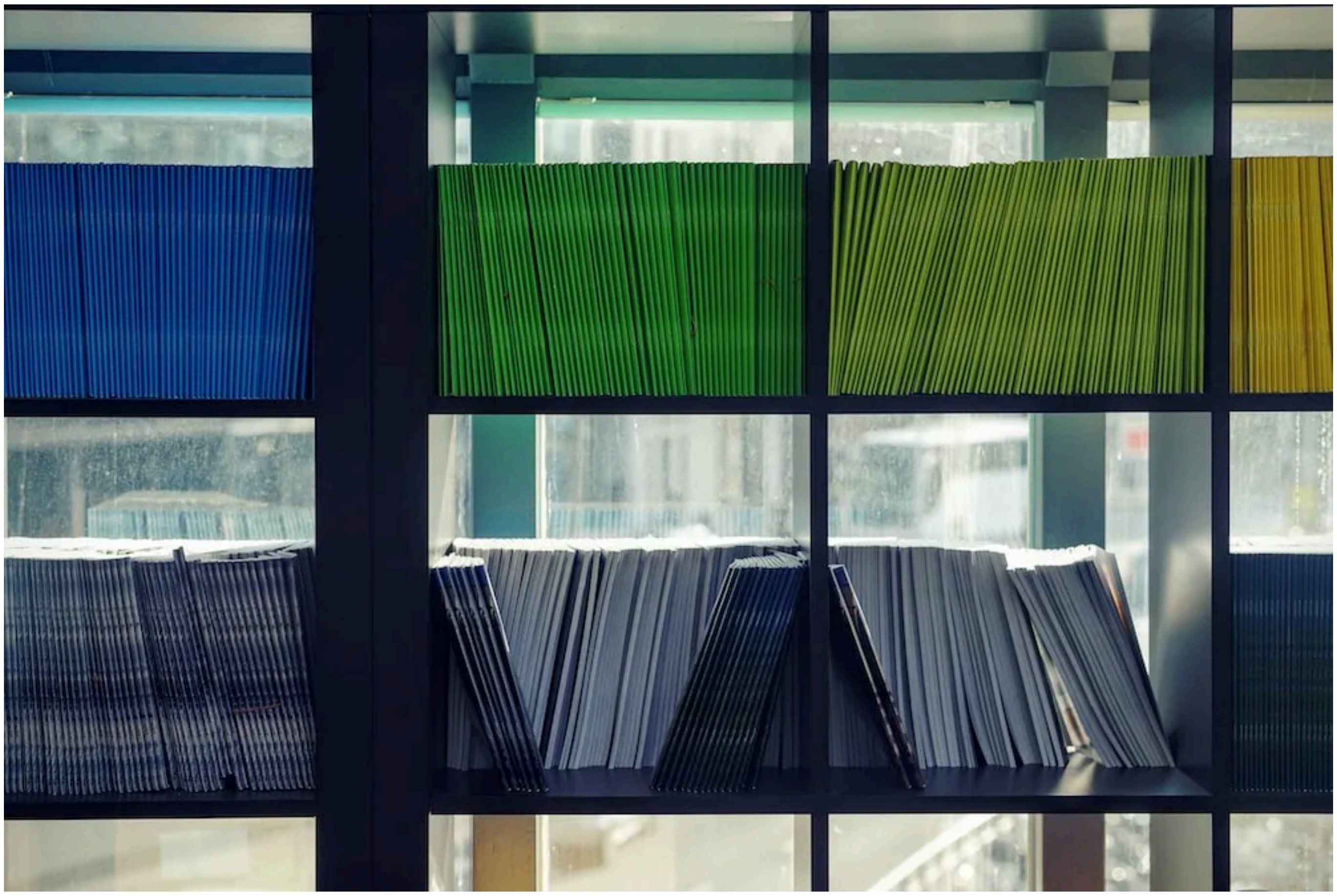
Image by Maarten van den Heuvel — Unsplash

Choosing the right data format is crucial in Data Science projects, impacting everything from data read/write speeds to memory consumption and interoperability. This article explores seven popular serialization/deserialization formats in Python, focusing on their speed and memory usage implications.

Through the analysis, we'll also see how we can use profiling in Python (using the `cProfile` built-in module) and how we can get statistics on memory usage for specific files in your filesystem (using the `os` Python module).

Of course, each project has its specificities, beyond just speed and memory usage. But we'll draw some trends, that can hopefully be useful to shed light on which format we can choose for a given project.

Understanding Serialization and Deserialization

> *Serialization is the process of saving an object (in Python, a pandas DataFrame for example) to a format that can be saved to a file for later retrieval. Deserialization is the reverse process.*

A dataframe is a Python object and cannot be persisted as is. It needs to be translated to a file to be able to load this object at a later stage.

When you save a dataframe, you “serialize” the data. And when you load it back, you “deserialize” or translate it back to a language-readable (here Python-readable) format.

Certain formats are widely used because they are human-readable, such as JSON or CSV. These two formats are also used because they are language agnostic. Just like protocol buffers, which were originally developed by Google. JSON and Protocol buffer are also popular for APIs and enable sending data between different services written in different languages.

On the other hand, some formats, like Python’s pickle, are language-specific and not ideal for transferring data between services in different programming languages. For example, for a machine learning use case, if a repository trains a model and serializes it in pickle, this file will only be able to be read from Python. So if the API that serves that machine learning model is written in Java, some transformations will have to be done before it can be used.

But for storing large data files, formats like CSV, JSON, and Protocol Buffers fall short in performance compared to more specialized formats like HDF5, Parquet, Feather or ORC.

When working with data in Python, pandas provide a convenient API to extract, transform and load data. Pandas supports a wide array of formats (all the supported ones can be found here), among which we chose the seven most used formats and compared their performance: JSON, CSV, Parquet, Pickle, Feather, HDF5, and ORC.

The code found to reproduce the analysis can be found here.

Experimental approach

Speed and **memory usage** are two key components to look at when choosing the right data format for your project. But these vary based on both the size of the data, and which type of data is being used.

To do the analysis, we created 3 datasets that should cover most use cases: a number only dataframe, a strings-only dataframe and a dataframe with both numbers and strings. Each dataframe is 10,000 rows over 100 columns and are defined by the following code:

```
df_numbers_only = pd.DataFrame(np.random.rand(10000, 100), columns=[f'col{i}' fo
df_strings_only = pd.DataFrame(np.random.choice(['a', 'b', 'c'], (10000, 100)), 
df_mixed = pd.DataFrame(np.random.choice(['a', 'b', 'c', np.random.rand()], (100
```

For the memory usage, we just look at what is the size of the dataset when saved. To measure memory usage, we'll use Python's `os` module, and specifically `os.stat`.

```
df_numbers_only.to_csv('df.csv')
file_stats = os.stat('df.csv')
print(file_stats)
print(f'File Size in Bytes is {file_stats.st_size}') #this is in Bytes
print(f'File Size in MegaBytes is {file_stats.st_size / (1024 * 1024)}') #here w
```

For the speed, we use Python's cProfile library to profile the time taken by functions to execute. Here, we take a window of 10 seconds and look at how many times the function executed. In essence:

```
def serialize_csv(df):
    """Serialize df to csv."""
    df.to_csv('test.csv')

func = serialize_csv
args = df_numbers_only

timer = pytool.time.Timer()
duration = datetime.timedelta(seconds=10)
with cProfile.Profile() as profile:
    while timer.elapsed < duration:
        func(*args)
    time_results[(df_name, file_format, operation)] = profile
```

Assessing speed: how fast are the different formats?

Delving straight into the heart of the analysis, let's look at the graphs created for evaluating serialization/deserialization speed. Note that we scale the time it took, so you can see on the graph below that csv with numbers only is the slowest process while using HDF5 for numbers only is the fastest, taking 1.09% (roughly 1/100) of the CSV one.

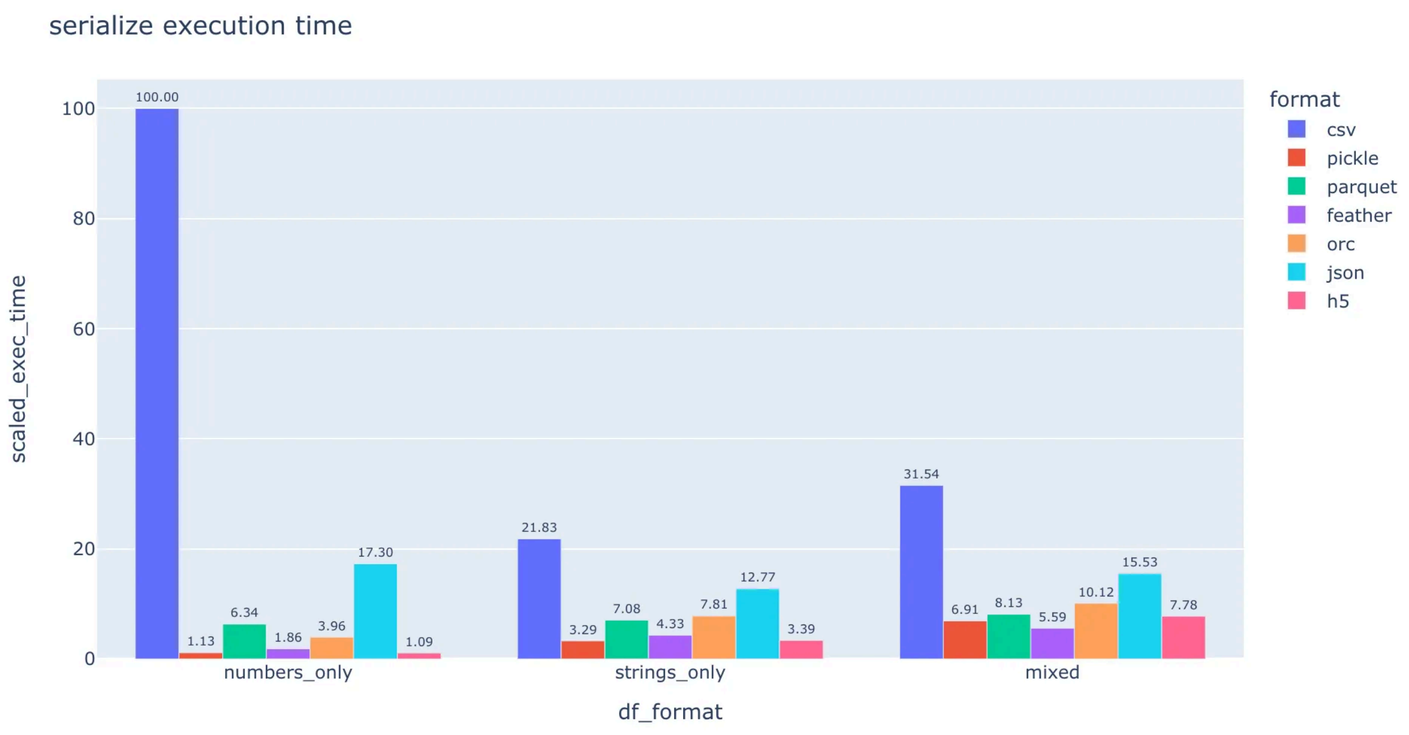

Image by author: save time by format

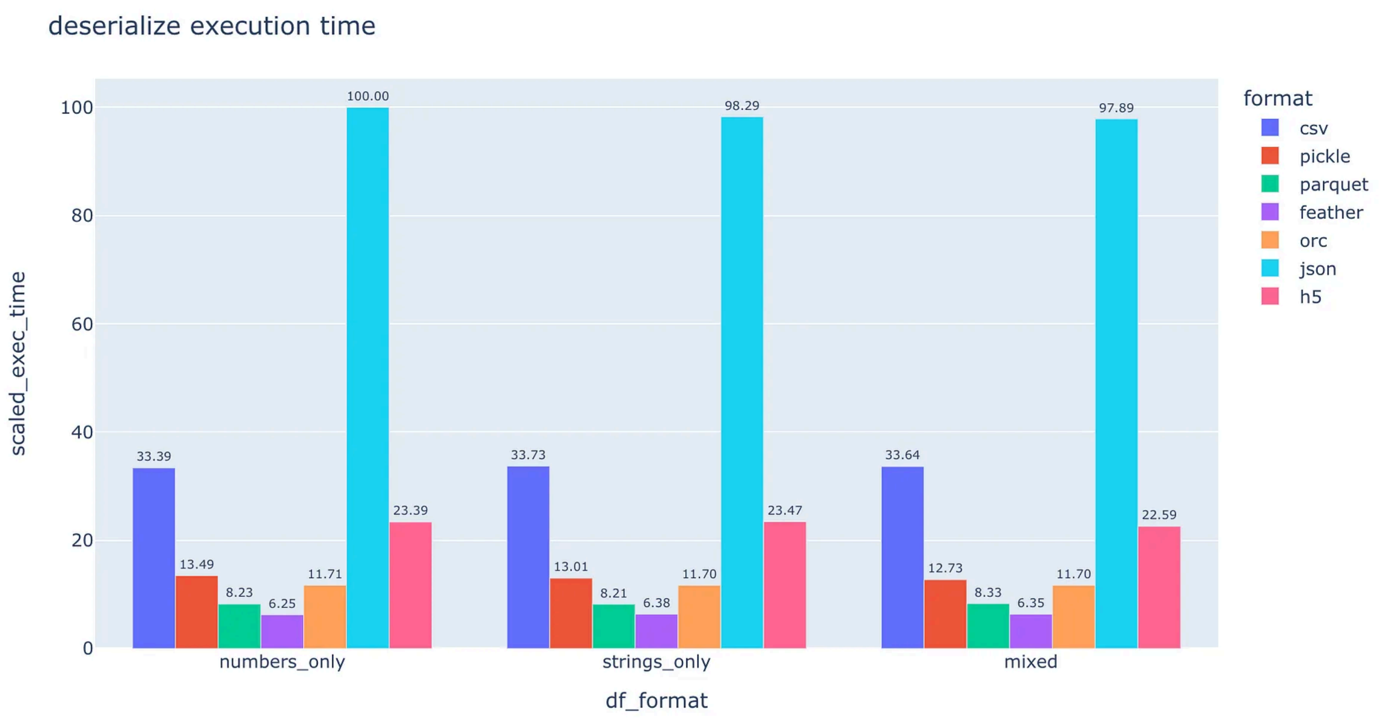

Image by author: load time by format

On the graphs above, we can observe a few interesting take aways in terms of speed:

- **Pickle/HDF5** are the fastest to save number-only data
- **Feather** is the fastest format to load the data.
- Human readable formats such as **JSON** or **CSV** are much slower than other formats, especially for saving/loading data that only contains numbers (almost 100x slower than pickle for instance)
- **HDF5** is interestingly very performant for saving data (on par with feather, parquet, pickle) but a bit less for loading data.

Memory usage: which format consumes less space?

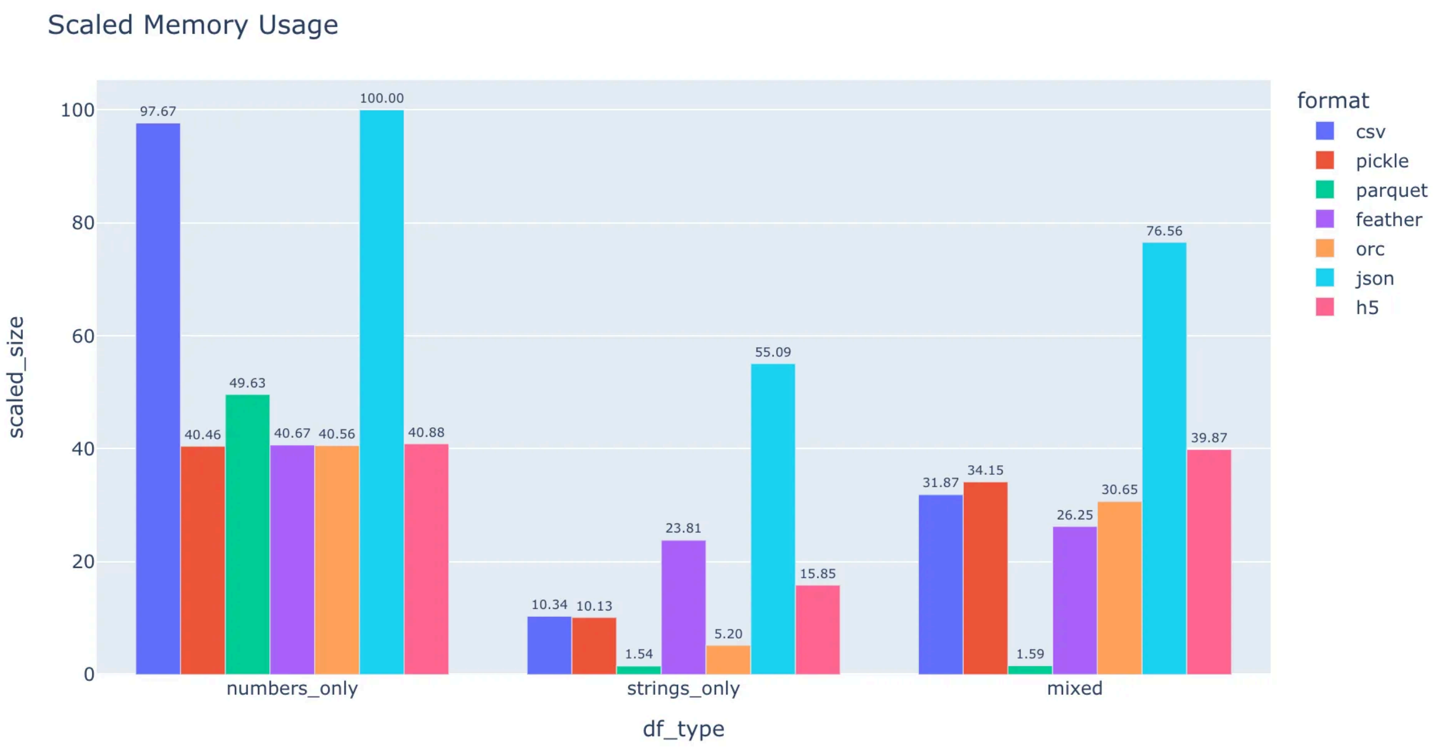

Image by author: scaled memory usage

We can observe that:

- once again, human-readable formats such as **CSV or JSON** are the least memory efficient formats. And JSON performs even worse than CSV.
- **parquet** is the most memory efficient format with the given size of the data (10,000x100), which makes sense given parquet is a column-oriented data format. You can read more about it here.
- once again, when dealing solely with numerical data, **pickle** is the most efficient format. Pickle being the native Python package for serialization, it is not surprising.
- while faster, **feather** data takes a lot more memory compared to parquet.

The findings can be summarized on the following table (with a bit less nuance than the analysis above):

Search this file…

Format	Best for	Speed (Serialization)	Speed (Deserialization)	Memory Usage
Pickle	Numerical data in Python projects	Fast	Average	Good for numbe
Parquet	Balancing speed and memory usage	Fast	Fast	Best
CSV	Human-readable data	Slow	Slow	Least Efficient
Feather	Speed over memory efficiency	Fast	Fast	Average
HDF5	Large numerical datasets	Fast	Average	Efficient
ORC	Optimized row columnar data	Average	Average	Efficient
JSON	Data interchange and readability	Slow	Slow	Least Efficient

dataformats.csv hosted with ❤️ by **GitHub** view raw

Gist by author

Conclusions

The right format always boils down to your project's unique demands. But we can still draw general conclusions from the above analysis. For example, unless interoperability is key and you need to visually see what the raw data looks like, you're better off not using human-readable formats such as **CSV or JSON**.

Overall, **parquet** seems to be a very solid choice as it is much better on the memory usage side of things while still being relatively fast.

It is also pivotal to understand the specificities of each format. For example, pickle is Python specific so you won't be able to read it from other languages. But for Python-centric projects, focused solely on numerical data handling, **pickle** seems to be the best choice (both in terms of speed and memory).

If you're managing a lot of data and want to minimize the amount of GB you're writing to disk, **parquet** is the way to go. But if speed is your primary focus, you might want to give **feather** a try but it will take more memory.

Long story short, only the specific requirements of your project will guide you towards the right format. But you can hopefully gain some insights from the above to get a grasp on which one is better in your specific case. Happy coding!

Data Programming Big Data Machine Learning Profiling

Data Science

Published in TDS Archive

Follow