Making Sense of Big Data

Do You Read Excel Files with Python? There is a 1000x Faster Way.

In this article, I'll show you five ways to load data in Python. Achieving a speedup of 3 orders of magnitude.

Source: https://www.hippopx.com/, public domain

As a Python user, I use excel files to load/store data as business people like to share data in excel or csv format. Unfortunately, Python is especially slow with Excel files.

In this article, I'll show you five ways to load data in Python. In the end, we'll achieve a speedup of 3 orders of magnitude. It'll be lightning-fast.

> Edit (18/07/2021): I found a way to make the process **5 times faster** (resulting in a 5000x speedup). I added it as a bonus at the end of the article.

Experimental Setup

Let's imagine that we want to load 10 Excel files with 20000 rows and 25 columns (that's around 70MB in total). This is a representative case where you want to load transactional data from an ERP (SAP) to Python to perform some analysis.

Let’s populate this dummy data and import the required libraries (we’ll discuss pickle and joblib later in the article).

```
import pandas as pd
import numpy as np
from joblib import Parallel, delayed
import time

for file_number in range(10):
    values = np.random.uniform(size=(20000,25))
```

```
import pandas as pd
import numpy as np
from joblib import Parallel, delayed
import time


for file_number in range(10):
 values = np.random.uniform(size=(20000,25))
 pd.DataFrame(values).to_csv(f”Dummy {file_number}.csv”)
 pd.DataFrame(values).to_excel(f”Dummy {file_number}.xlsx”)
 pd.DataFrame(values).to_pickle(f”Dummy
{file_number}.pickle”)
```

5 Ways to Load Data in Python

Idea #1: Load an Excel File in Python

Let’s start with a straightforward way to load these files. We’ll create a first Pandas Dataframe and then append each Excel file to it.

```
start = time.time()
df = pd.read_excel(“Dummy 0.xlsx”)
for file_number in range(1,10):
 df.append(pd.read_excel(f”Dummy {file_number}.xlsx”))
end = time.time()
print(“Excel:”, end — start)
```

```
>> Excel: 53.4
```

```python
start = time.time()
df = pd.read_excel("Dummy 0.xlsx")
for file_number in range(1,10):
    df.append(pd.read_excel(f"Dummy {file_number}.xlsx"))
end = time.time()
```

A simple way to import Excel files in Python.

It takes around 50 seconds to run. Pretty slow.

Idea #2: Use CSVs rather than Excel Files

Let’s now imagine that we saved these files as .csv (rather than .xlsx) from our ERP/System/SAP.

```
start = time.time()
df = pd.read_csv(“Dummy 0.csv”)
for file_number in range(1,10):
 df.append(pd.read_csv(f”Dummy {file_number}.csv”))
end = time.time()
print(“CSV:”, end — start)
```

```
>> CSV: 0.632
```

```python
start = time.time()
df = pd.read_csv("Dummy 0.csv")
for file_number in range(1,10):
    df.append(pd.read_csv(f"Dummy {file_number}.csv"))
end = time.time()
```

Importing csv files in Python is 100x faster than Excel files.

We can now load these files in 0.63 seconds. That’s nearly 10 times faster!

Python loads CSV files 100 times faster than Excel files. Use CSVs.

Con: csv files are nearly always bigger than .xlsx files. In this example .csv files are 9.5MB, whereas .xlsx are 6.4MB.

Idea #3: Smarter Pandas DataFrames Creation

We can speed up our process by changing the way we create our pandas DataFrames. Instead of appending each file to an existing DataFrame,

1. We load each DataFrame independently in a list.

2. Then concatenate the whole list in a single DataFrame.

```
start = time.time()
df = []
for file_number in range(10):
 temp = pd.read_csv(f"Dummy {file_number}.csv")
 df.append(temp)
df = pd.concat(df, ignore_index=True)
end = time.time()
print("CSV2:", end — start)
```

```
>> CSV2: 0.619
```

```
start = time.time()
df = []
for file_number in range(10):
    temp = pd.read_csv(f"Dummy {file_number}.csv")
    df.append(temp)
df = pd.concat(df, ignore_index=True)
```

A smarter way to import csv files in Python

We reduced the time by a few percent. Based on my experience, this trick will become useful when you deal with bigger Dataframes (df >> 100MB).

Idea #4: Parallelize CSV Imports with Joblib

We want to load 10 files in Python. Instead of loading each file **one by one**, why not loading them all, at once, in parallel?

We can do this easily using joblib.

```
start = time.time()
def loop(file_number):
 return pd.read_csv(f"Dummy {file_number}.csv")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)
(file_number) for file_number in range(10))
df = pd.concat(df, ignore_index=True)
```

```
end = time.time()
print("CSV//:", end — start)

>> CSV//: 0.386
```

```
start = time.time()
def loop(file_number):
    return pd.read_csv(f"Dummy {file_number}.csv")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)(file_num
df = pd.concat(df, ignore_index=True)
```

Import CSV files in Python in Parallel using Joblib.

That's nearly twice as fast as the single core version. However, as a general rule, do not expect to speed up your processes eightfold by using 8 cores (here, I got x2 speed up by using 8 cores on a Mac Air using the new M1 chip).

Simple Paralellization in Python with Joblib

Joblib is a simple Python library that allows you to run a function in //. In practice, joblib works as a list comprehension. Except each iteration is performed by a different thread. Here's an example.

```
def loop(file_number):
 return pd.read_csv(f"Dummy {file_number}.csv")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)
(file_number) for file_number in range(10))

#equivalent to
df = [loop(file_number) for file_number in range(10)]
```

```
def loop(file_number):
    return pd.read_csv(f"Dummy {file_number}.csv")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)(file_num

#equivalent to
```

Think as joblib as a smart list comprehension.

Idea #5: Use Pickle Files

You can go (much) faster by storing data in pickle files—a specific format used by Python—rather than .csv files.

Con: you won't be able to manually open a pickle file and see what's in it.

```
start = time.time()
def loop(file_number):
 return pd.read_pickle(f"Dummy {file_number}.pickle")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)
(file_number) for file_number in range(10))
df = pd.concat(df, ignore_index=True)
end = time.time()
print("Pickle//:", end — start)
```

```
>> Pickle//: 0.072
```

```
start = time.time()
def loop(file_number):
    return pd.read_pickle(f"Dummy {file_number}.pickle")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)(file_num
df = pd.concat(df, ignore_index=True)
```

We just cut the running time by 80%!

In general, it is much faster to work with pickle files than csv files. But, on the other hand, pickles files usually take more space on your drive (not in this specific example).

In practice, you will not be able to extract data from a system directly in pickle files.

I would advise using pickles in the two following cases:

1. You want to save data from one of your Python processes (and you don't plan on opening it on Excel) to use it later/in another process. Save your Dataframes as pickles instead of .csv

2. You need to reload the same file(s) multiple times. The first time you open a file, save it as a pickle so that you will be able to load the pickle version directly next time.
   Example: Imagine that you use transactional monthly data (each month you load a new month of data). You can save all historical

data as .pickle and, each time you receive a new file, you can load it once as a .csv and then keep it as a .pickle for the next time.

Bonus: Loading Excel Files in Parallel

Let's imagine that you received excel files and that you have no other choice but to load them as is. You can also use joblib to parallelize this. Compared to our pickle code from above, we **only** need to update the loop function.

```
start = time.time()
def loop(file_number):
    return pd.read_excel(f"Dummy {file_number}.xlsx")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)
(file_number) for file_number in range(10))
df = pd.concat(df, ignore_index=True)
end = time.time()
print("Excel//:", end - start)


>> 13.45
```

```
start = time.time()
def loop(file_number):
    return pd.read_excel(f"Dummy {file_number}.xlsx")
df = Parallel(n_jobs=-1, verbose=10)(delayed(loop)(file_num
df = pd.concat(df, ignore_index=True)
```

How to load excel files using parallelization in Python.

We could reduce the loading time by 70% (from 50 seconds to 13 seconds).

You can also use this loop to create pickle files on the fly. So that, next time you load these files, you'll be able to achieve lightning fast loading times.

```
def loop(file_number):
    temp = pd.read_excel(f"Dummy {file_number}.xlsx")
    temp.to_pickle(f"Dummy {file_number}.pickle")
    return temp
```

Recap

By loading pickle files in parallel, we decreased the loading time from 50 seconds to less than a tenth of a second.

- Excel: 50 seconds
- CSV: 0.63 seconds
- Smarter CSV: 0.62 seconds
- CSV in //: 0.34 seconds
- Pickle in //: 0.07 seconds
- Excel in //: 13.5 seconds

Bonus #2: 4x Faster Parallelization

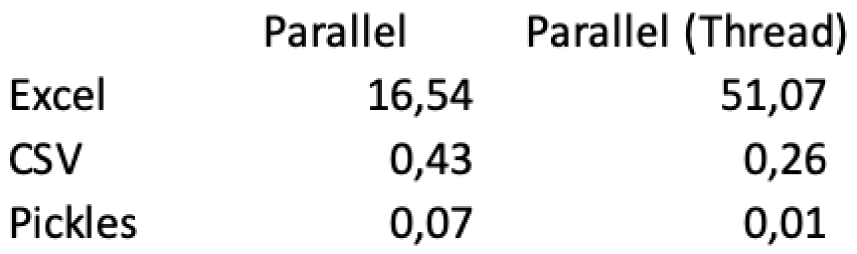
Joblib allows to change the parallelization backend to remove some overheads. You can do this by giving *prefer=”threads"* to *Parallel*.

```python
def loop(file_number):
    return pd.read_pickle(f"Dummy {file_number}.pickle")
df = Parallel(n_jobs=-1, verbose=0, prefer="threads")(del
df = pd.concat(df, ignore_index=True)
```

Using prefer="threads" will allow you to run your process even faster.

We obtain a speed of around 0.0096 seconds (over 50 runs with a 2021 MacBook Air).

Using prefer=”threads” with CSV and Excel parallelization gives the following results.



	Parallel	Parallel (Thread)
Excel	16,54	51,07
CSV	0,43	0,26
Pickles	0,07	0,01

As you can see using the “Thread” backend results in a worse score when reading Excel files. But to an astonishing performance with pickles (it takes 50 seconds to load Excel files one by one, and only 0.01 seconds to load the data reading pickles files in //).

. . .

☞ Let's connect on LinkedIn!

About the Author

Nicolas Vandeput is a supply chain data scientist specialized in demand forecasting and inventory optimization. He founded his consultancy company SupChains in 2016 and co-founded SKU Science —a fast, simple, and affordable demand forecasting platform—in 2018. Passionate about education, Nicolas is both an avid learner and enjoys teaching at universities: he has taught forecasting and inventory optimization to master students since 2014 in Brussels, Belgium. Since 2020 he is also teaching both subjects at CentraleSupelec, Paris, France. He published *Data Science for Supply Chain Forecasting* in 2018 (2nd edition in 2021) and *Inventory Optimization: Models and Simulations* in 2020.

— 2 BOOKS —
AVAILABLE ON
amazon
NEW EDITION !
DATA SCIENCE FOR SUPPLY CHAIN
INVENTORY OPTIMIZATION MODELS AND SIMULATIONS
WWW.SUPCHAINS.COM