Using SAS to create pandas data

Ian D. Gow

20 January 2026

1 Introduction

A strong point of pandas is its expressiveness. Its API allows users to explore data using succinct and (generally) intuitive code. However, some of this expressiveness relies on data being in forms (for example, with dates ready to serve as an index) that often differ from the data we have, and pandas can struggle to manipulate the data into those forms, especially with larger data sets.

SAS might be another approach to manipulating data for pandas. My Python package `wrds2pg` offers a `sas_to_pandas()` function that can run code on the WRDS server and return the results as a pandas dataframe. While not quite as fast as using Ibis with the PostgreSQL server, SAS performs pretty well with this task.

> **Tip**
>
> The following command (run in the terminal on your computer) installs the packages you need.
>
> ```
> pip install wrds2pg --upgrade
> pip install pandas
> ```
>
> The code assumes you have set the environment variable `WRDS_ID` to your WRDS ID.
> This note was written using Quarto. The source code for this note is available here and the latest version of this PDF is here.

2 Expressive pandas

Since 2012, pandas has become the leading **data frame library** in Python. A real strength of pandas appears to be its expressiveness, which allows a user to explore data with succinct code. To show this, I will adapt an example from Hilpisch (2019). The following code reads data from a GitHub page ...

```
import pandas as pd

url = ("https://raw.githubusercontent.com/yhilpisch/"
       "py4fi2nd/refs/heads/master/source/"
       "tr_eikon_eod_data.csv")

ticker_list = ["AAPL.O", "MSFT.O", "INTC.O", "AMZN.O", "GS.N"]

data = (pd
    .read_csv(url, index_col=0, parse_dates=True)
    [ticker_list]
)
```

... and then one line of code generates Figure 1.

```
data.plot(figsize=(8, 8), subplots=True);
```

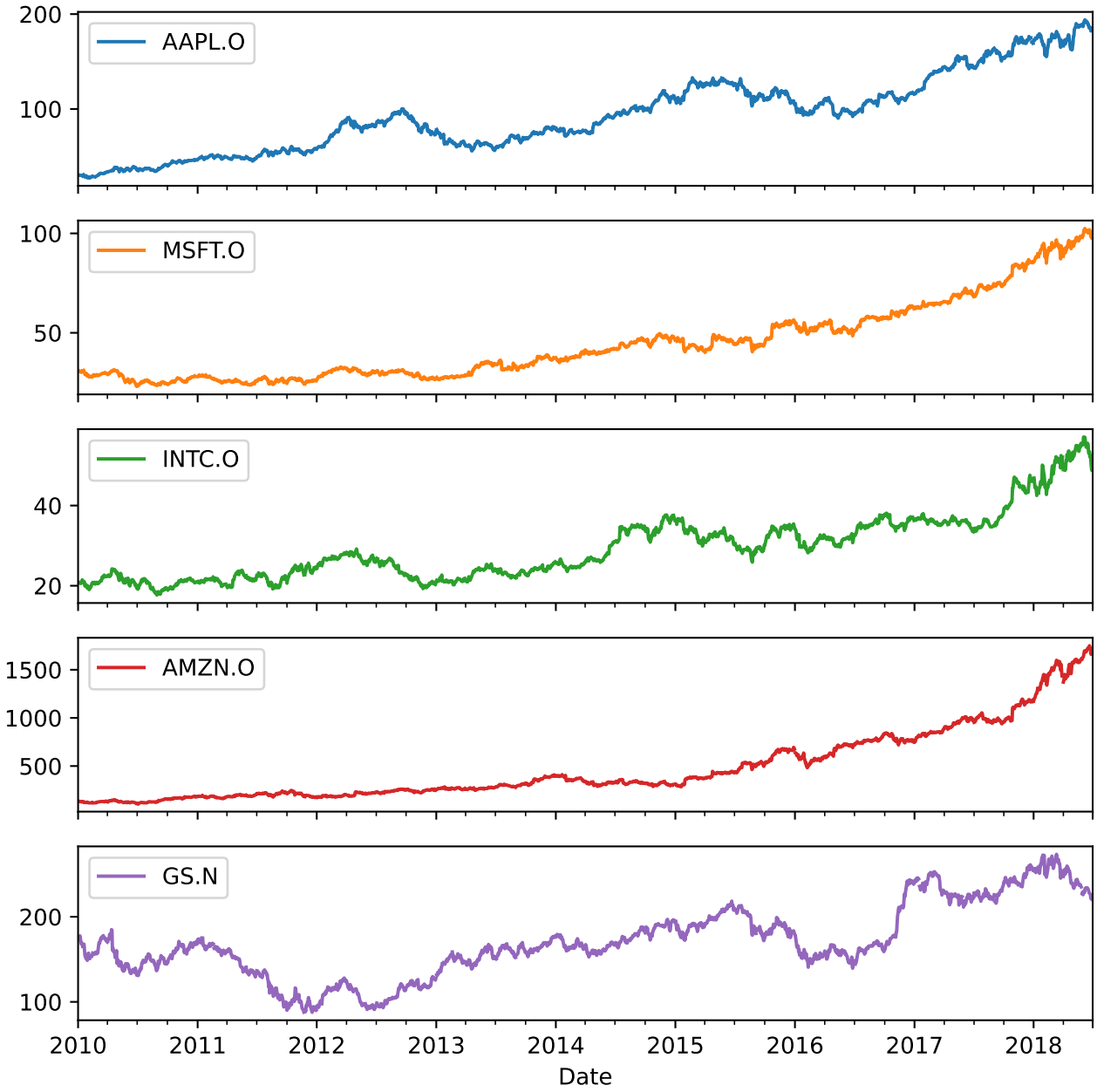

Figure 1: Stock prices for five firms: Original

Looking at the code, it seems that pandas has magically intuited that the data set comprises a number of time series, so a call to the `.plot()` method of the `pd.DataFrame` generates a plot and `subplots=True` makes a subplot for each series. Of course, it wasn't some special instinct for the meaning of data that allowed pandas to do this. Rather, by having dates in the first column of the CSV and then telling pandas to use that column to generate the `Index` for the `pd.DataFrame`, we get the data in the following

form:

```
data.head()
```

Date	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N
2010-01-01	NaN	NaN	NaN	NaN	NaN
2010-01-04	30.572827	30.950	20.88	133.90	173.08
2010-01-05	30.625684	30.960	20.87	134.69	176.14
2010-01-06	30.138541	30.770	20.80	132.25	174.26
2010-01-07	30.082827	30.452	20.60	130.00	177.67

As can be seen, `Date` is different from the other "columns" of the data frame; in a sense, it's not a column at all, but the index for the data frame:

```
data.index
```

```
DatetimeIndex(['2010-01-01', '2010-01-04', '2010-01-05', '2010-01-06',
               '2010-01-07', '2010-01-08', '2010-01-11', '2010-01-12',
               '2010-01-13', '2010-01-14',
               ...
               '2018-06-18', '2018-06-19', '2018-06-20', '2018-06-21',
               '2018-06-22', '2018-06-25', '2018-06-26', '2018-06-27',
               '2018-06-28', '2018-06-29'],
              dtype='datetime64[ns]', name='Date', length=2216, freq=None)
```

Having the data in this form allows us to access the data with succinct code. We can use `.loc[]` to select by date ...

```
data.loc['2010-01-08']
```

```
AAPL.O     30.282827
MSFT.O     30.660000
INTC.O     20.830000
AMZN.O    133.520000
GS.N      174.310000
Name: 2010-01-08 00:00:00, dtype: float64
```

... and `[]` to select by column.

```
data['AAPL.O']
```

```
Date
2010-01-01           NaN
2010-01-04     30.572827
2010-01-05     30.625684
2010-01-06     30.138541
2010-01-07     30.082827
                  ...
2018-06-25    182.170000
2018-06-26    184.430000
2018-06-27    184.160000
2018-06-28    185.500000
2018-06-29    185.110000
Name: AAPL.O, Length: 2216, dtype: float64
```

An important operation for financial time series is **resampling** (Hilpisch, 2019, p. 215). For example, we could transform the daily data in `data` into weekly data with one line:

```
data.resample('W').last().head()
```

Date	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N
2010-01-03	NaN	NaN	NaN	NaN	NaN
2010-01-10	30.282827	30.66	20.83	133.52	174.31
2010-01-17	29.418542	30.86	20.80	127.14	165.21
2010-01-24	28.249972	28.96	19.91	121.43	154.12
2010-01-31	27.437544	28.18	19.40	125.41	148.72

Similarly with monthly data ...

```
data.resample('ME').last().head()
```

Date	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N
2010-01-31	27.437544	28.1800	19.40	125.41	148.72
2010-02-28	29.231399	28.6700	20.53	118.40	156.35
2010-03-31	33.571395	29.2875	22.29	135.77	170.63

Date	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N
2010-04-30	37.298534	30.5350	22.84	137.10	145.20
2010-05-31	36.697106	25.8000	21.42	125.46	144.26

These examples demonstrate the expressiveness of pandas, especially with financial time series data. This facility is less surprising once you realize that pandas began life when Wes McKinney was working at AQR Capital Management.

However, I think there is a danger of overestimating the facility of working with pandas from such examples. My experience is that data sets rarely come in a form that allows one to use a `DatetimeIndex` with series identifiers as the columns.

Many data sets have a wider range of data types (one of the reasons for creating pandas in the first place) and data are often provided in a form that needs work to get to something like `data` above.

For example, many researchers, including academics, generally get stock prices and returns for US firms from CRSP. According to its website, "the Center for Research in Security Prices, LLC (CRSP) maintains the most comprehensive collection of security price, return, and volume data for the NYSE, AMEX and NASDAQ stock markets." (See Chapter 7 of Gow and Ding (2024) for more on CRSP.) Academic researchers generally get CRSP data through Wharton Research Data Services, more commonly referred to as WRDS (pronounced "words").

So a question might be: What do we need to do to get CRSP data into the form above? Can I use SAS?

The answer to the second question is "yes" and the rest of the note provides an answer to the first question.

3 Generating Figure 1 using SAS

I put the SAS code shown in Listing 1 and Listing 2 in a file `dsf_to_pd.sas` and I can load it as follows.

```
from pathlib import Path

sas_code = Path("dsf_to_pd.sas").read_text(encoding="utf-8")
```

I will use `sas_to_pandas()` from `wrds2pg` to run the SAS code on the WRDS server.

```
from wrds2pg import sas_to_pandas
```

The first step is to look up tickers and link them to the security identifier used by CRSP (`permno`). The tickers found in the data provided by Hilpisch above include suffixes that indicate the exchange the stocks traded on.

An important detail about tickers is that they get reused over time. So a ticker match with a `permno` may have a range of dates over which it is valid. I'm going to assume that the tickers in Hilpisch (2019) were valid on the last date observed in his data set and look for the PERMNO match valid on that date. This part is performed using `PROC SQL` to create `tickers` from `crsp.stocknames`.

Having obtained the `permno` data, I can use this to get the relevant return data from `crsp.dsf`.

Again I use `PROC SQL` to get stock prices (`prc`) and returns both with (`ret`) and without (`retx`) dividends and other distributions from `crsp.dsf` and store these in `dsf_sub`, which is sorted by `ticker` and `date`. It turns out that the price series used by Hilpisch (2019) did not include the effects of distributions, so I use `retx` in the analysis below.

The next step is to recreate, as best we can, the price series used in Hilpisch (2019). My assumption is that the prices are adjusted for splits such that the adjusted price at the end of the series used by Hilpisch (2019) equals the unadjusted price on that date (i.e., what should be in `prc`). I then recreate preceding adjusted prices in each time series by working back from the ending price using returns (and I confirmed by inspecting the data in Hilpisch (2019) that `retx` is the appropriate return measure).

The first data set is `dsf_g`, which cumulates `(1 + retx)` by `ticker` as the variable `growth`. Then `lastvals` grabs the last values for each `ticker` and stores these as `prc_last` and `growth_last`.

One way to interpret part of the SAS below that creates `dsf_adj` is, taking one stock at a time, that `growth_last` is the cumulative returns for each stock over the whole time series and the adjusted price for each date is the final price (`prc_last`) multiplied by the cumulative returns to date (`growth`) divided by `growth_last`.

The final step of the SAS code dumps the contents of `dsf_adj` to CSV so it can be read into pandas. Note that the dates do not get parsed by the current version of `sas_to_pandas()`, so we will have to do that ourselves in pandas below.

```
%%ptime
dsf_adj = sas_to_pandas(sas_code)
```

Wall time: 4.839 s

Once we have a `pd.DataFrame`, we convert `date` to a `datetime64[ns]` and set the index using the resulting column.

```
%%ptime
dsf_adj["date"] = pd.to_datetime(dsf_adj["date"], format="%Y%m%d")
```

Wall time: 4.63 ms

Next, we `.pivot()` the data (in the earlier note we did this using polars or Ibis).

```
%%ptime
data_alt = (dsf_adj
    .pivot(index="date", columns="ticker", values="prc_adj")
)
```

Wall time: 5.79 ms

I use `clean_tickers()` to order the columns in the data frame so that Figure 2 better mirrors Figure 1.

```
import re

clean_tickers = [re.sub(r"\.[A-Z]+$", "", t) for t in ticker_list]
```

It turns out that the index in the original data frame is filled out with empty rows on public holidays, likely because the original data included commodities and exchange rates that traded on those dates and we omitted those data here. Having the dates in the index actually makes the plot look better, so I effectively add them to the index of `data_alt` by using `.reindex(data.index)`.

```
%%ptime
data_alt = data_alt.reindex(data.index)[clean_tickers]
```

Wall time: 0.64 ms

```
data_alt.plot(figsize=(8, 8), subplots=True);
```

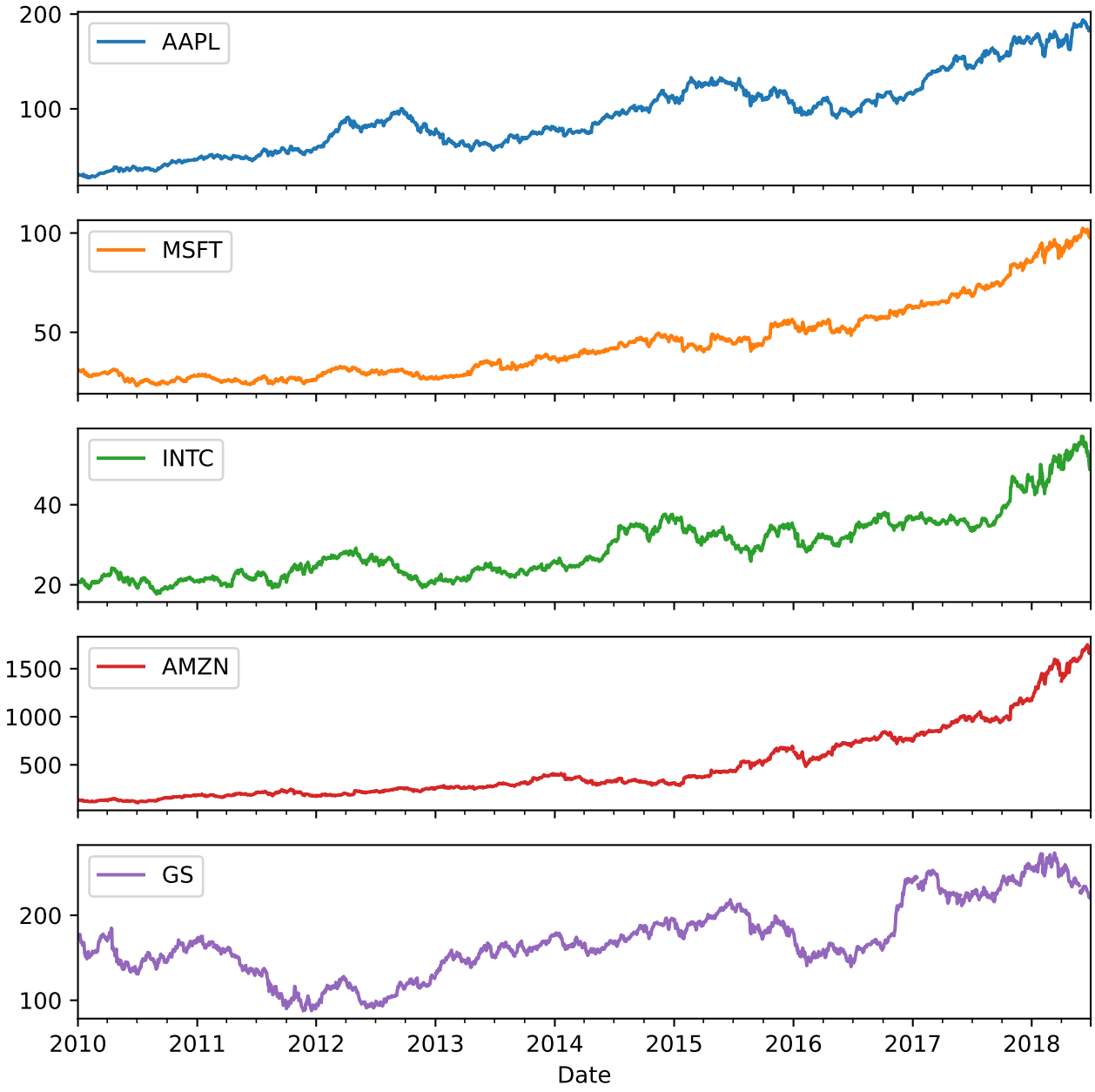

Figure 2: Stock prices for five firms: Using SAS and `wrds2pg`

References

Gow, I.D., Ding, T., 2024. Empirical research in accounting: Tools and methods. Chapman & Hall/CRC, London, UK. https://doi.org/10.1201/9781003456230

Hilpisch, Y.J., 2019. Python for finance: Mastering data-driven finance, 2nd ed. O'Reilly Media, Sebastopol, CA.

Listing 1 SAS code found in `dsf_to_pd.sas` (Part 1)

```
%let start_date = '01JAN2010'd;
%let end_date   = '29JUN2018'd;

proc sql;
  create table tickers as
  select permno, ticker
  from crsp.stocknames as s
  where ticker in ('AAPL', 'MSFT', 'INTC', 'AMZN', 'GS') and
    &end_date between namedt and nameenddt;
quit;

proc sql;
  create table dsf_sub as
  select
      t.ticker, d.date, d.prc, d.ret, d.retx
  from crsp.dsf as d
  inner join work.tickers as t
    on d.permno = t.permno
  where d.date between &start_date and &end_date
  order by t.ticker, d.date;
quit;

/* Pass 1: compute cumulative growth */
data dsf_g;
  set dsf_sub;
  by ticker;

  retain growth;
  if first.ticker then growth = 1;
  else growth = growth * (1 + retx);
run;
```

Listing 2 SAS code found in `dsf_to_pd.sas` (Part 2)

```
/* Pass 2: grab the last prc and last growth for each ticker */
data lastvals(keep=ticker prc_last growth_last);
  set dsf_g;
  by ticker;
  if last.ticker then do;
    prc_last = prc;
    growth_last = growth;
    output;
  end;
run;

/* Join back and compute adjusted price */
proc sql;
  create table dsf_adj as
  select
      g.ticker,
      g.date,
      /* adjusted price */
      (l.prc_last * g.growth / l.growth_last) as prc_adj
  from dsf_g as g
  inner join lastvals as l
    on g.ticker = l.ticker
  order by g.ticker, g.date;
quit;

proc export data=dsf_adj outfile=stdout dbms=csv;
run;
```