Empirical Research in Accounting with Python (parquet version)

Ian D. Gow*

24 December 2023

Recent developments in Python led me to do some research into the possibility of translating *Empirical Research in Accounting: Tools and Methods* from R to Python. As discussed below, I think it would be possible to do this without completely losing the elegance and efficiency of the current approach. Nonetheless, a significant amount of work would be required and this could not happen before 2024. In the meantime, I recommend Python users interested in our book to consider learning a little R.

The code used to produce this note can be found here. It should be straightforward to compile this document using RStudio.

Our book *Empirical Research in Accounting: Tools and Methods* incorporates an extensive amount of code for a few reasons. One reason is that one's understanding of a paper can be much deeper if one can get one's "hands dirty" with the analysis in a paper, including identifying and executing variants on the analysis in the paper. Another reason is that a core goal is to provide readers with the data analysis skills needed to do their own research.

Another book taking a similar approach, but perhaps more focused on the coding, is *Tidy Finance with R*. Since *Tidy Finance with R* was released earlier in 2023, the author team has added a fourth member to create a Python version of the book, which can be found on the book's website.

Suppressing some messy details, translation of either *Empirical Research in Accounting: Tools and Methods* or *Tidy Finance with R* from R to Python using standard tools like SQLAchemy and Pandas involves two main steps.

1. Translating `dbplyr` code that operates on remote data frames to SQL.
2. Translating `dplyr` code that operates on local data frames to Pandas code.[1]

*University of Melbourne, ian.gow@unimelb.edu.au

[1]The package `dbplyr` provides a set of verbs for interacting with remote data sources, while `dplyr` provides more or less the same verbs for interacting with tibbles and other local data frames.

With *Tidy Finance with R*, almost all of the first step is concentrated in Part II of their book, "Financial Data". While the data sets created in Part II are stored in an SQLite database, the remainder of the book uses SQLite largely as a simple data store.[2] All analyses involve loading entire tables from SQLite and processing the data using `dplyr`.

In contrast, the approach in *Empirical Research in Accounting: Tools and Methods* uses `dbplyr` throughout. So the first step of this approach to a Python translation of the current book would mean that SQL strings appeared throughout the book. While it would be possible to move to the data-management approach used by the Tidy Finance team, this would involve costs. First, the chapters would no longer be independent. Some code would depend on data sets created in a portion of the book largely dedicated to extracting data from the WRDS database. Second, users would need to think about managing a local database. While not a huge task, *Empirical Research in Accounting: Tools and Methods* does not ask users to maintain *any* persistent data storage. Third, there can be real benefits to executing data analysis steps in a database. For example, the code here that creates the `risk_asymmetry` data frame takes about one second to run, but would take many minutes using an approach that loaded data into a data frame and ran regressions in R or Python.[3]

Beyond these reasons, there are additional complications with moving *Empirical Research in Accounting: Tools and Methods* to Python. First, we assume very little of our readers. While the much shorter *Tidy Finance with R* elects to dive in head-first, we have a chapter providing a tutorial on R and another chapter introducing users to regression analysis using R. These chapters would require more than a simple code translation. Second, our book leans heavily on the companion R package `farr` for data sets and functions and we would have to work out how to replicate elements of this in a Python version.

Given our desire to retain the core approach used in *Empirical Research in Accounting: Tools and Methods*, translation of our book to Python has had to wait for a more `dbplyr`-like approach to emerge. (This is apart from the fact that we still need to finish the R version!) I recently noted that Wes McKinney, the creator of Pandas, has joined Posit, the firm once known as RStudio. In his blog, he discusses Ibis, "a lazily-evaluated expression system that is pleasant to use, extensible, and can support a broad set of SQL-like and non-SQL-like data processing use cases."

As outlined below, I think it is broadly feasible to move to a Python-based approach without completely forgoing the elegance and performance of the R-based approach. While a Tidyverse user will find that some things are missing in Python with Ibis, these don't seem to be deal-breakers. For example, Python doesn't have pipes (`|>` or `%>%` in R), but the chaining of methods gives something similar. Also, Python lacks the **non-standard evaluation** goodies provided by R.

[2]Little would change with *Tidy Finance with R* if the data tables were instead stored as individual `.rds` data files.
[3]Earlier versions of the code in fact took many minutes to run.

1 Translating from R to Python: A case study

In this note, I translate code from the `dbplyr` form it takes in the **parquet-focused version** of *Empirical Research in Accounting: Tools and Methods* into equivalent Ibis code. The code I chose for translation uses `crsp.dsf`, a relatively large table that illustrates nicely the benefits of working with remote data.[4]

Apart from importing Ibis, I also set `ibis.options.interactive` so that displayed output from data frames is limited to ten rows.

```
import ibis
ibis.options.interactive = True
```

On my computers, I store the location of my parquet files in an environment variable `DATA_DIR`.

```
from os.path import join
data_dir = "/Users/igow/Dropbox/pq_data"
```

Now I connect to the the local parquet data via DuckDB and establish variables representing `crsp.dsf` and `crsp.stocknames`. Note that these are effectively **remote** data frames, as no data is brought into memory. As such these lines take almost no time to run.

```
con = ibis.duckdb.connect()

dsf = con.read_parquet(join(data_dir, "crsp", "dsf.parquet"))
stocknames = con.read_parquet(join(data_dir, "crsp", "stocknames.parquet"))
```

Now that we have our remote data frames set up, we can begin to interrogate the data. First we ask how many rows are in the `crsp.dsf` table.

```
dsf.count().to_pandas()
```

```
105258380
```

[4]The careful reader will notice that the R code in the book uses parquet data. However a version that used a PostgreSQL database would be almost identical. In itself, this nicely illustrates a key benefit of an approach using `dplyr` and `dbplyr`.

Ibis takes a different approach from `dbplyr` in translating code to SQL. In `dbplyr`, many commonly used functions available to `dplyr` are translated to SQL equivalents in the dialect matching the connection supplied. For example `year(date)` is translated to `EXTRACT(year FROM "date")` in DuckDB's dialect of SQL. If no translation is found, then the function is passed along for the SQL query engine to interpret. An example of this is seen in `date_part('year', date)` below.

```
library(dplyr, warn.conflicts = FALSE)
library(DBI)
library(reticulate)

con <- dbConnect(RPostgres::Postgres())

dsf <- tbl(con, Id(table = "dsf", schema = "crsp"))

dsf |>
  mutate(year = year(date),
         year_alt = date_part('year', date)) |>
  select(permno, date, year, year_alt) |>
  show_query()
```

```
<SQL>
SELECT
  "permno",
  "date",
  EXTRACT(year FROM "date") AS "year",
  date_part('year', "date") AS "year_alt"
FROM "crsp"."dsf"
```

It seems that Ibis takes a different approach that is more like the second of these two options in most cases. However, Ibis requires that we register each function that we want to use as a user-defined function (UDF). We will want to use `date_part()` below, so we register this function as follows. We import `udf` and then add a **decorator** before our function to let Python know that we are registering a built-in scalar UDF. Because the function is handled by DuckDB, the body of our function is simply an ellipsis (`...`). All we have to do is indicate the slots for the arguments (`field` and `source` in this case) and returned data type using the type hint `-> int`.

```
from ibis import udf

@udf.scalar.builtin
```

```
def date_part(field, source) -> int:
    ...
```

Below we will use other UDFs and we register these now.

```
@udf.scalar.builtin
def regexp_matches(string, pattern) -> bool:
    ...

@udf.scalar.builtin
def coalesce(x, y) -> float:
    ...

@udf.scalar.builtin
def exp(x) -> float:
    ...

@udf.scalar.builtin
def ln(x) -> float:
    ...
```

While the following code is unnecessary (Ibis does this translation without us having to ask for it), it does illustrate how one can register aggregate functions such as `sum()` and `avg()`.

```
@ibis.udf.agg.builtin
def sum(x: float) -> float:
    ...
```

Now we get the count of observations on `crsp.dsf` by year using the following code. We retrieve the (small) result set as a Pandas data frame. We sort this by descending value of `year` so that the most recent rows come first.

```
dsf. \
    mutate(year=date_part("year", dsf.date)). \
    group_by("year"). \
    aggregate(n=dsf.count()). \
    order_by([ibis.desc("year")]). \
    to_pandas()
```

```
    year        n
0   2023  2353668
```

```
1   2022  2389874
2   2021  2187044
3   2020  1948489
4   2019  1911581
..   ...      ...
94  1929   201537
95  1928   180698
96  1927   172364
97  1926   160937
98  1925      509

[99 rows x 2 columns]
```

We will next take a subset of rows where `date` is 7 January 1986 and retrieve the (fairly small) result set as a Pandas data frame. This will allow users to see an important difference between Ibis/Pandas and `dbplyr`/`dplyr`.

In R, we might select just the fields we want before using `collect()` to create a local data frame, as here:

```
dsf_subset <-
  dsf |>
  filter(date == "1986-01-07") |>
  select(permno, date, ret) |>
  collect()
```

Alternatively, we might collect all the data, and just select the ones we need from the local data frame. In the vast majority of cases, one can apply the same methods one can apply to a local data frame (e.g., a "tibble") as one can apply to a remote data frame.

```
dsf_subset <-
  dsf |>
  filter(date == "1986-01-07") |>
  collect() |>
  select(permno, date, ret)
```

However, things are a bit more complicated with Ibis/Pandas. There is no `select()` method applicable to a Pandas data frame. Instead, would use `[["permno", "date", "ret"]]` to indicate that we want just those three columns. While `[["permno", "date", "ret"]]` does work with a remote data frame, it means mixing in Pandas-style methods with more dplyr-like methods.[5]

```
dsf_subset = dsf. \
  filter(dsf.date == "1986-01-07"). \
  select("permno", "date", "ret"). \
  to_pyarrow(). \
  to_pandas()

dsf_subset[["permno", "date", "ret"]]
```

```
      permno        date       ret
0      10000  1986-01-07       NaN
1      10015  1986-01-07  0.000000
2      10031  1986-01-07  0.000000
3      10057  1986-01-07  0.026549
4      10065  1986-01-07  0.006410
...      ...         ...       ...
6424   93252  1986-01-07 -0.019608
6425   93279  1986-01-07  0.000000
6426   93287  1986-01-07  0.000000
6427   93308  1986-01-07  0.000000
6428   93316  1986-01-07  0.012987

[6429 rows x 3 columns]
```

Note that permno is a floating-point type in the WRDS PostgreSQL database, though it really should be an integer.[6]

Now, we will use crsp.stocknames to look up the PERMNO for Apple so that we can make a plot of Apple's stock performance over time.

```
apple_permno = stocknames. \
    filter(regexp_matches(stocknames.comnam, "^APPLE COM")). \
    select("permno"). \
    to_pandas(). \
    permno[0]
```

[5]Remove the .to_pandas() from the code to check this. Given the line select("permno", "date", "ret") prior to to_pandas(), the [["permno", "date", "ret"]] attached to the last line is not doing anything of consequence. Try removing that portion of the code, or the select("permno", "date", "ret") line to see the effects of doing so.

[6]WRDS is currently working on a project to clean up data types in its database.

From this we learn that Apple's PERMNO is 14593. I use this to apply `filter()` to `crsp.dsf` to get just those rows applicable to Apple, calculate the natural logarithm of gross return (I use `coalesce()` to set missing returns—if any—to zero). I then `sum()` these returns over the window `w`. Applying the aggregate function `sum()` in a window context turns `sum()` into a window function. The window here partitions the data by `permno` (like `dplyr`, Ibis overloads `group_by()` for this purpose) and sets the window to run all the way up the current row (while we give an argument to `following`, the argument to `preceding` is not supplied and is therefore the default of "all rows preceding" in the window).

Note that we import and use the special variable `_`. In effect, this indicates the current data frame and allows us to refer to a column created in previous steps and not available in the underlying table (`crsp.dsf` in this case).

```
from ibis import _

w = ibis.window(group_by="permno", following=0)

plot_data = dsf. \
    filter(dsf.permno == apple_permno). \
    mutate(log_ret = ln(1 + coalesce(dsf.ret, 0))). \
    mutate(sum_ret = exp(_.log_ret.sum().over(w))). \
    select("permno", "date", "ret", "sum_ret"). \
    to_pyarrow(). \
    to_pandas()
```

With `plot_data` in hand, we can make our plot, which can be seen in Figure 1.

```
import matplotlib.pyplot as plt
plt.plot(plot_data.date, plot_data.sum_ret);
plt.show()
```

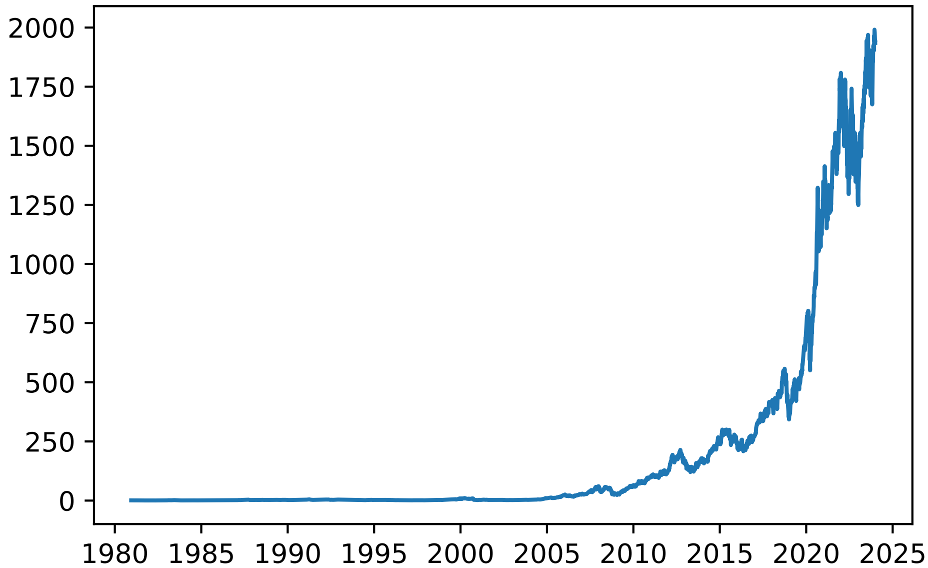

Figure 1: Stock performance of Apple