Photo by Twitter: @jankolario on Unsplash

# Turn Your Excel Workbook Into a SQLite Database

Transforming spreadsheets into queryable database tables

Scott A. Adams · Follow

Published in Towards Data Science · 8 min read · May 18, 2020

147 2

## Introduction

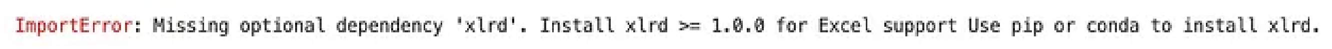
A **relational database** is a collection of data tables — sets of rows and columns that store individual pieces of data — that can be connected to each other. In this way, a relational database is not totally dissimilar from an Excel workbook with related datasets stored across multiple worksheets. With that thought in mind, this post moves through an example using Python to transform an Excel spreadsheet into a database that can be queried using **Structured Query Language (SQL)**.

## Data

The example in this post uses data from the *Superstore-Sales* dataset, which can be found here. This dataset is stored as an Excel workbook and contains example sales transaction data stored across the following four sheets: *sales, orders, customers,* and *products.*

### Reading The Data In Python

First, if you have not installed the `xlrd` package using `conda` or `pip`, do so before starting a Python session, or else you will encounter the following error when trying to read in the Excel file (even with *pandas* installed).



```
ImportError: Missing optional dependency 'xlrd'. Install xlrd >= 1.0.0 for Excel support Use pip or conda to install xlrd.
```

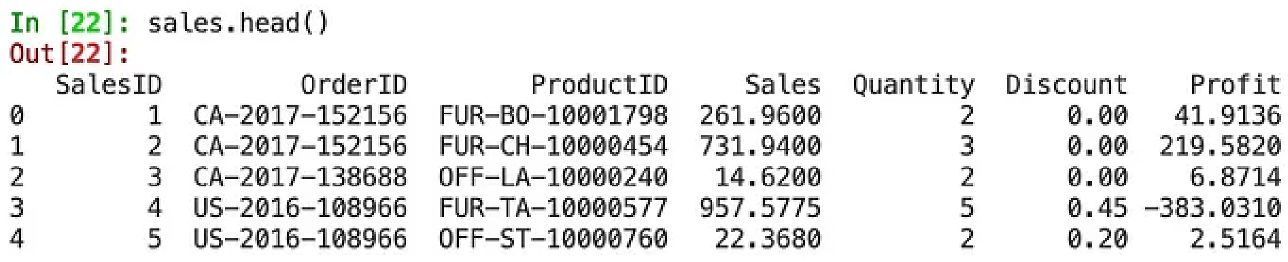
Now let's start a Python session and import *pandas* and *sqlite3* using `import pandas` and `import sqlite3`. We will read the data from each Excel spreadsheet into a separate *pandas* data frame using `read_excel`, as shown in the code below.

```python
sales = pd.read_excel(
    'data/Superstore_4sheets.xlsx',
    sheet_name='sales',
    header=0)

orders = pd.read_excel(
    'data/Superstore_4sheets.xlsx',
    sheet_name='orders',
    header=0)

customers = pd.read_excel(
    'data/Superstore_4sheets.xlsx',
    sheet_name='customers',
    header=0)

products = pd.read_excel(
    'data/Superstore_4sheets.xlsx',
    sheet_name='products',
    header=0)
```

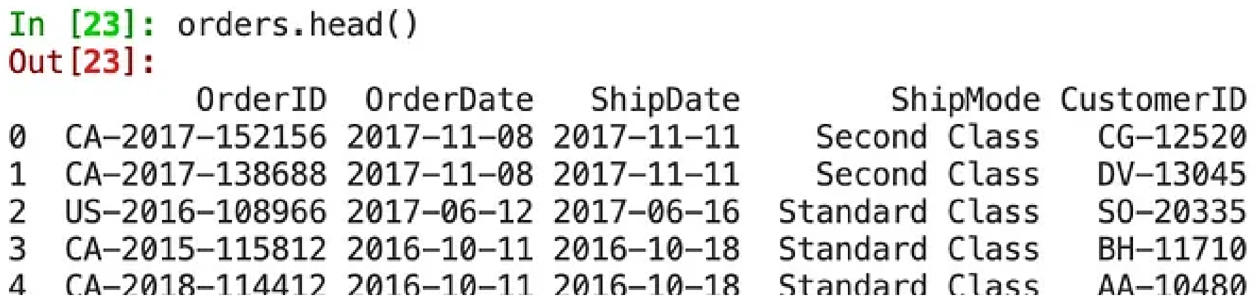
**read_superstore.py** hosted with ❤ by **GitHub** **view raw**

In each code chunk, the first parameter in `read_excel` identifies the name of the file to be processed. If necessary, include the path in which the file resides. For example, when I ran this code my data was stored in the `data` directory, but my working directory was one directory up. The second parameter, `sheet_name =` , specifies the name of the spreadsheet in the workbook to be processed. The last parameter, `header=0` , tells Python that the first row in the spreadsheet being read contains the column headers. Remember that Python is zero-indexed, which is why we use `0` and not `1` to identify the first row. By explicitly identifying the header row, the values in the first row of each spreadsheet will be treated as the column names for its respective data frame.

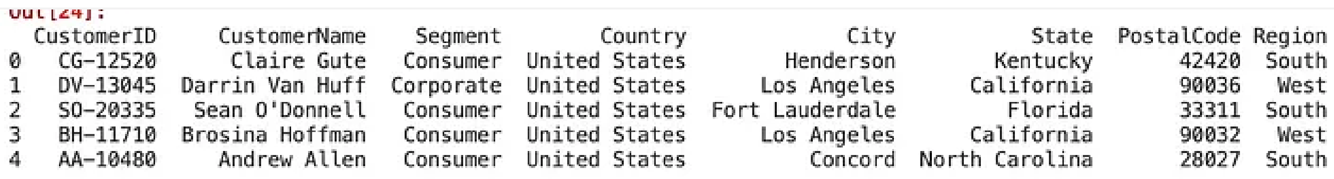
Let’s take a look at the first few rows of each data frame just to make sure everything looks correct.



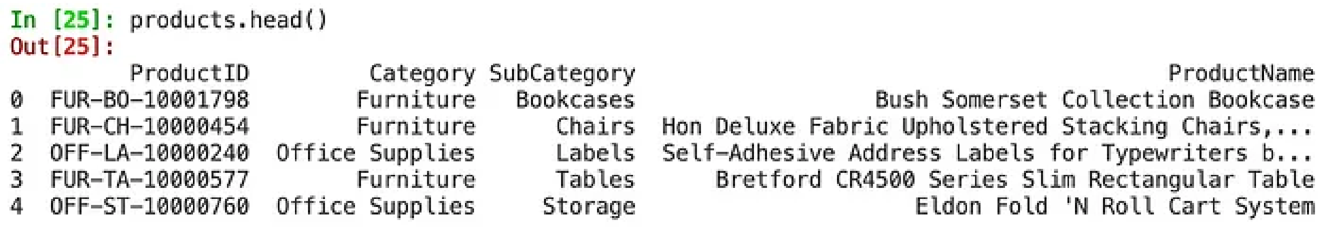
```
In [22]: sales.head()
Out[22]:
   SalesID         OrderID        ProductID     Sales  Quantity  Discount    Profit
0        1  CA-2017-152156  FUR-BO-10001798  261.9600         2      0.00   41.9136
1        2  CA-2017-152156  FUR-CH-10000454  731.9400         3      0.00  219.5820
2        3  CA-2017-138688  OFF-LA-10000240   14.6200         2      0.00    6.8714
3        4  US-2016-108966  FUR-TA-10000577  957.5775         5      0.45 -383.0310
4        5  US-2016-108966  OFF-ST-10000760   22.3680         2      0.20    2.5164
```



```
In [23]: orders.head()
Out[23]:
          OrderID  OrderDate   ShipDate        ShipMode CustomerID
0  CA-2017-152156 2017-11-08 2017-11-11    Second Class   CG-12520
1  CA-2017-138688 2017-11-08 2017-11-11    Second Class   DV-13045
2  US-2016-108966 2017-06-12 2017-06-16  Standard Class   SO-20335
3  CA-2015-115812 2016-10-11 2016-10-18  Standard Class   BH-11710
4  CA-2018-114412 2016-10-11 2016-10-18  Standard Class   AA-10480
```



```
Out[24]:
  CustomerID     CustomerName    Segment        Country            City           State  PostalCode Region
0   CG-12520      Claire Gute   Consumer  United States       Henderson        Kentucky       42420  South
1   DV-13045  Darrin Van Huff  Corporate  United States     Los Angeles      California       90036   West
2   SO-20335   Sean O'Donnell   Consumer  United States  Fort Lauderdale         Florida       33311  South
3   BH-11710  Brosina Hoffman   Consumer  United States     Los Angeles      California       90032   West
4   AA-10480     Andrew Allen   Consumer  United States         Concord  North Carolina       28027  South
```



```
In [25]: products.head()
Out[25]:
         ProductID         Category SubCategory                                        ProductName
0  FUR-BO-10001798        Furniture   Bookcases                  Bush Somerset Collection Bookcase
1  FUR-CH-10000454        Furniture      Chairs  Hon Deluxe Fabric Upholstered Stacking Chairs,...
2  OFF-LA-10000240  Office Supplies      Labels  Self-Adhesive Address Labels for Typewriters b...
3  FUR-TA-10000577        Furniture      Tables      Bretford CR4500 Series Slim Rectangular Table
4  OFF-ST-10000760  Office Supplies     Storage                     Eldon Fold 'N Roll Cart System
```

Great! Now let's make that database.

## Creating a SQLite Database

There are multiple relational database products available and the specific product we will use here is **SQLite**. This is a lightweight SQL database engine that can be used to create databases stored as files on one's personal computer. We can initiate a new SQLite database connection object and assign this object to a variable. Below, I named this variable `db_conn`.

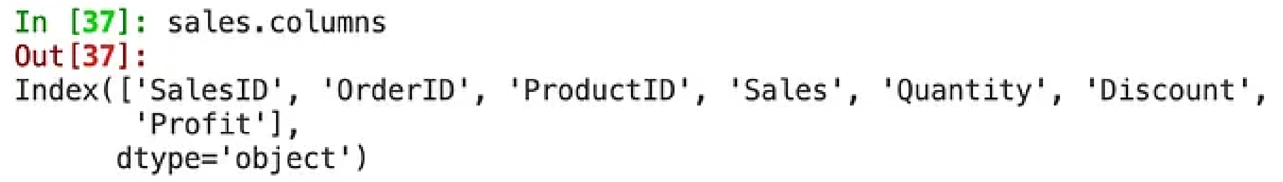
```
db_conn = sqlite3.connect("data/superstore.db")
```

When this code above is executed, a new file named *superstore.db* will be created in the *data* directory (assuming *superstore.db* does not already exist in this directory). The newly created *superstore.db* file is an empty SQLite database at this point (i.e., it has no tables). `db_conn` is also pointing to the *superstore.db* database and can be thought of as an interface for working with this database. Cool! But wait…how do we actually run the SQL code that will create the data tables? Glad you asked. We now need to establish a **cursor** object, which is a tool that executes SQL code against the database of interest. Here, the cursor is assigned to a variable I named `c`.

```
c = db_conn.cursor()
```

The next step is to create the tables that will be included in the database, though it is important to emphasize that this next step of creating the tables will result in empty tables. Later on, we will populate our tables with the data from the four previously-created data frames. Before we move further, however, let's take a look at the actual SQL code that will be used to create the *sales* table to better understand the table creation process.

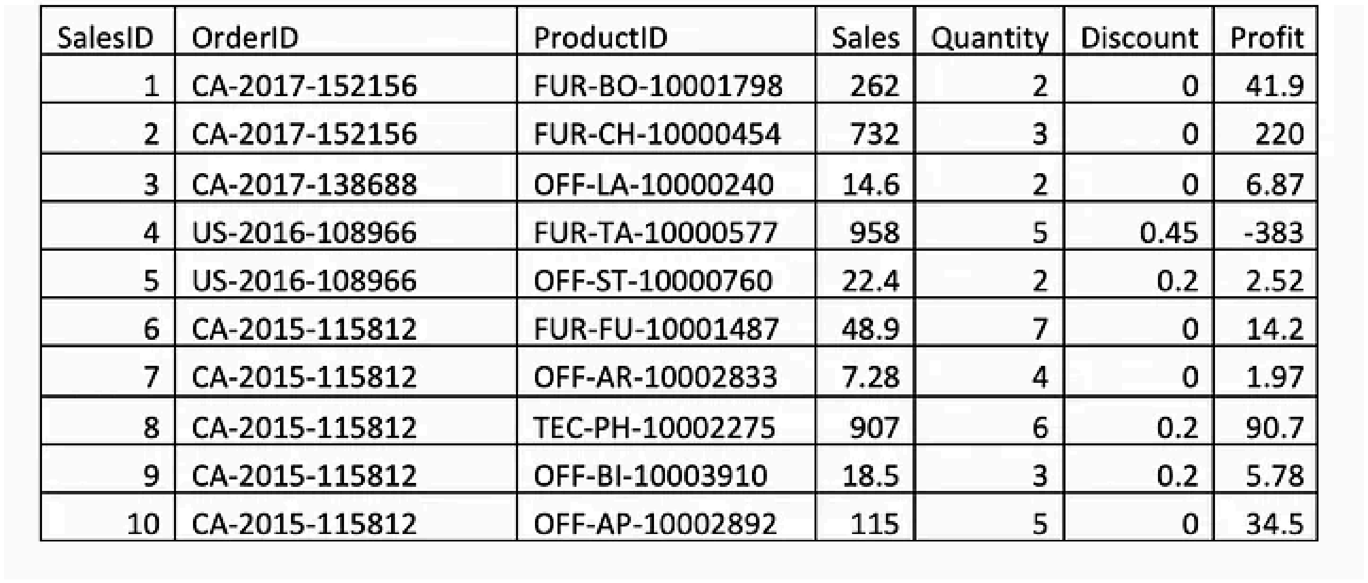
Line #1 in the code above provides a command to create a table named *sales* and lines #2–8 create 7 new columns in *sales*: SalesID, OrderID, ProductID, Sales, Quantity, Discount, and Profit. The data type for each column is specified to the right of the respective column name. More information on SQLite datatypes can be found here. Note that the order the columns matches the order of the columns from the associated data frame.



```
In [37]: sales.columns
Out[37]:
Index(['SalesID', 'OrderID', 'ProductID', 'Sales', 'Quantity', 'Discount',
       'Profit'],
      dtype='object')
```

The ordering of the columns in the `CREATE TABLE` statement is intentional as this ensures that the appropriate values from the data frame go to the intended column in the database table. For example, had I made OrderID the first column and SalesID the second, SalesID values would be written to the OrderID column in the *sales* table and the SalesID column in this table would contain OrderID values.

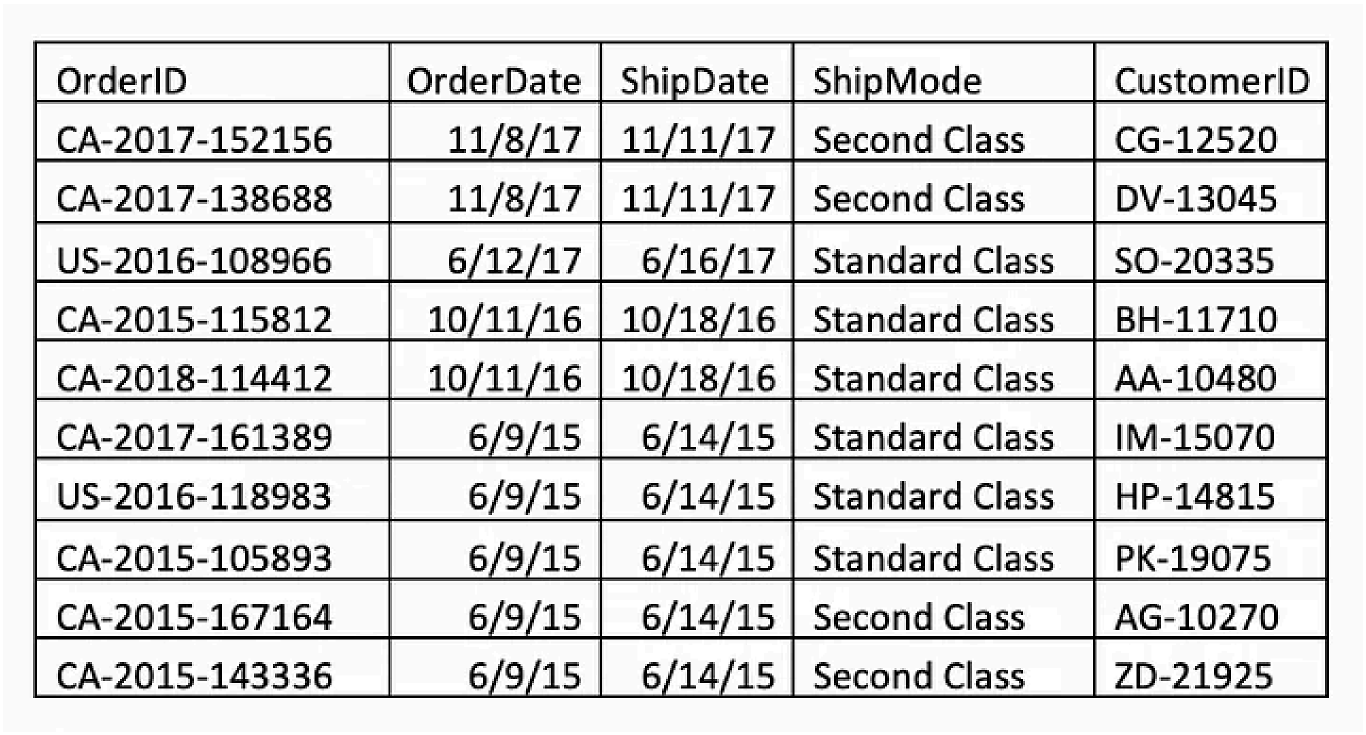
Line #9 establishes the table's **primary key**, which is a column that contains values that uniquely identify each row. In the *sales* table, SalesID satisfies the requirements for a primary key because no two rows have the same SalesID value.



| SalesID | OrderID | ProductID | Sales | Quantity | Discount | Profit |
|---|---|---|---|---|---|---|
| 1 | CA-2017-152156 | FUR-BO-10001798 | 262 | 2 | 0 | 41.9 |
| 2 | CA-2017-152156 | FUR-CH-10000454 | 732 | 3 | 0 | 220 |
| 3 | CA-2017-138688 | OFF-LA-10000240 | 14.6 | 2 | 0 | 6.87 |
| 4 | US-2016-108966 | FUR-TA-10000577 | 958 | 5 | 0.45 | -383 |
| 5 | US-2016-108966 | OFF-ST-10000760 | 22.4 | 2 | 0.2 | 2.52 |
| 6 | CA-2015-115812 | FUR-FU-10001487 | 48.9 | 7 | 0 | 14.2 |
| 7 | CA-2015-115812 | OFF-AR-10002833 | 7.28 | 4 | 0 | 1.97 |
| 8 | CA-2015-115812 | TEC-PH-10002275 | 907 | 6 | 0.2 | 90.7 |
| 9 | CA-2015-115812 | OFF-BI-10003910 | 18.5 | 3 | 0.2 | 5.78 |
| 10 | CA-2015-115812 | OFF-AP-10002892 | 115 | 5 | 0 | 34.5 |

Snapshot of the "sales" table

Line #10 in the previous set of code establishes the first of two **foreign keys** for *sales*. A foreign key is a column in one table that is the primary key in another table. For instance, notice that different rows can share the same OrderID value in the *sales* table, which disqualifies OrderID from being the primary key in this table. However, in the *orders* table each row does contain a unique OrderID value. As such, OrderID can serve as the primary key for *orders*.

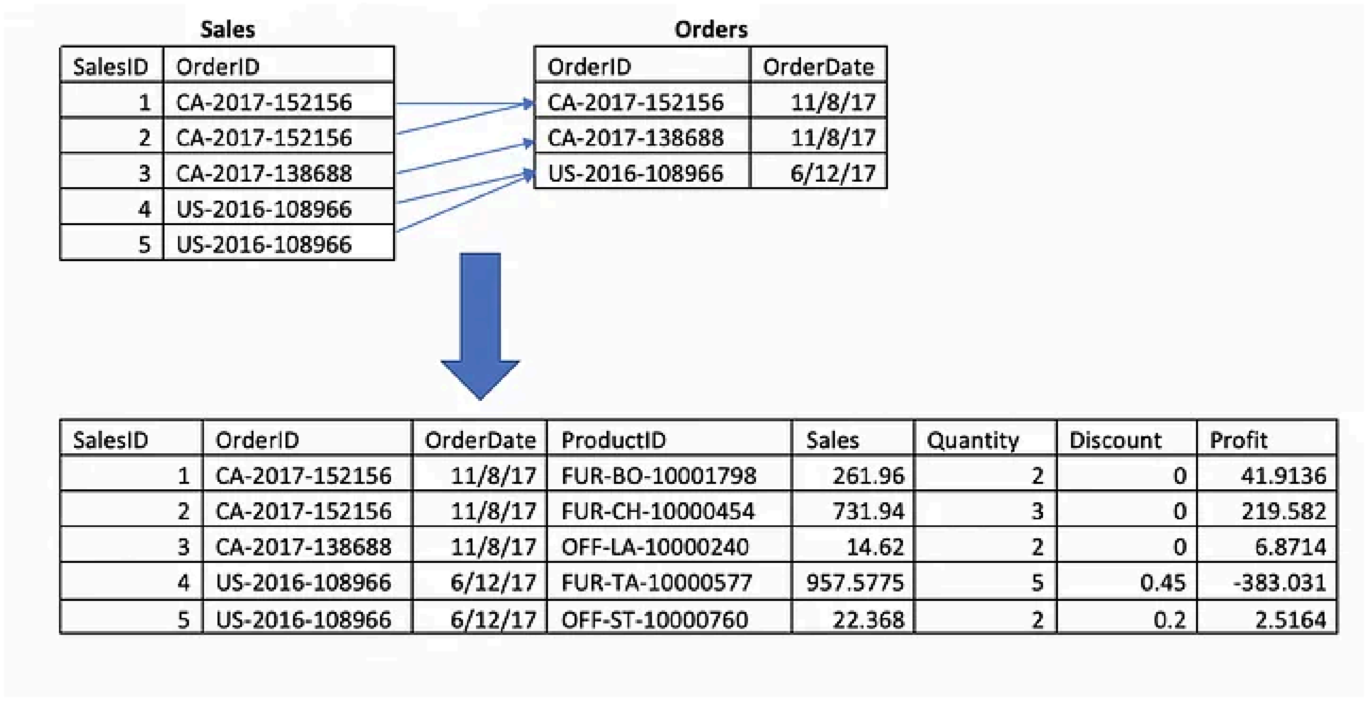


| OrderID | OrderDate | ShipDate | ShipMode | CustomerID |
|---|---|---|---|---|
| CA-2017-152156 | 11/8/17 | 11/11/17 | Second Class | CG-12520 |
| CA-2017-138688 | 11/8/17 | 11/11/17 | Second Class | DV-13045 |
| US-2016-108966 | 6/12/17 | 6/16/17 | Standard Class | SO-20335 |
| CA-2015-115812 | 10/11/16 | 10/18/16 | Standard Class | BH-11710 |
| CA-2018-114412 | 10/11/16 | 10/18/16 | Standard Class | AA-10480 |
| CA-2017-161389 | 6/9/15 | 6/14/15 | Standard Class | IM-15070 |
| US-2016-118983 | 6/9/15 | 6/14/15 | Standard Class | HP-14815 |
| CA-2015-105893 | 6/9/15 | 6/14/15 | Standard Class | PK-19075 |
| CA-2015-167164 | 6/9/15 | 6/14/15 | Second Class | AG-10270 |
| CA-2015-143336 | 6/9/15 | 6/14/15 | Second Class | ZD-21925 |

Snapshot of the "orders" table

Let's return to the actual code in line #10 in the previous `CREATE TABLE` statement. The first part of this code, `FOREIGN KEY(OrderID)` , establishes that the OrderID column in *sales* is a foreign key. The second part, `REFERENCE orders(OrderID)` then specifies the table and primary key to which OrderID refers. Line #11 from the `CREATE TABLE` statement follows the same logic for the ProductID column.

By specifying the primary and foreign keys, we are able to create a "map" that shows how the tables in the database relate to each other. For example, someone who is familiar with SQL concepts but unfamiliar with this particular data could take a look at the `CREATE TABLE` statement that creates the *sales* table and recognize that data from the *sales* and *orders* tables can be brought together, or **joined**, by matching on values in the OrderID columns from both tables. To briefly illustrate how joining works, consider how we can add OrderDate to the *sales* table by "joining" on OrderID, as illustrated below.



Sales

| SalesID | OrderID |
|---|---|
| 1 | CA-2017-152156 |
| 2 | CA-2017-152156 |
| 3 | CA-2017-138688 |
| 4 | US-2016-108966 |
| 5 | US-2016-108966 |

Orders

| OrderID | OrderDate |
|---|---|
| CA-2017-152156 | 11/8/17 |
| CA-2017-138688 | 11/8/17 |
| US-2016-108966 | 6/12/17 |

| SalesID | OrderID | OrderDate | ProductID | Sales | Quantity | Discount | Profit |
|---|---|---|---|---|---|---|---|
| 1 | CA-2017-152156 | 11/8/17 | FUR-BO-10001798 | 261.96 | 2 | 0 | 41.9136 |
| 2 | CA-2017-152156 | 11/8/17 | FUR-CH-10000454 | 731.94 | 3 | 0 | 219.582 |
| 3 | CA-2017-138688 | 11/8/17 | OFF-LA-10000240 | 14.62 | 2 | 0 | 6.8714 |
| 4 | US-2016-108966 | 6/12/17 | FUR-TA-10000577 | 957.5775 | 5 | 0.45 | -383.031 |
| 5 | US-2016-108966 | 6/12/17 | OFF-ST-10000760 | 22.368 | 2 | 0.2 | 2.5164 |

A basic visual demonstration of how joining works

## Finishing the Database

With a basic overview of the table creation process established, let's create all the database tables. This is where we get to put the cursor, `c` to work. We can run `c.execute()` and include the desired SQL code as a string in the parentheses to run said SQL code against the database to which we are currently connected (i.e., `data/superstore.db`).

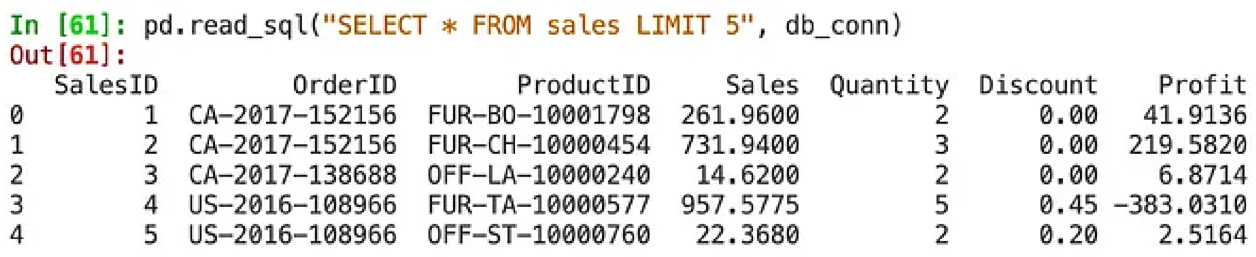
It is now time to populate the tables in the database with the relevant data from the four data frames we previously created. Fortunately, this step can be accomplished with ease using *pandas*' `to_sql` on each data frame (see more details on `to_sql` here). The code in the block below transfers the data from each of the four data frames into the appropriate tables in the database.

The first parameter in each line of code above identifies the name of the database table to which the values from the data frame will be written and the following parameter specifies the actual database connection. Next, `if_exists='append'` tells `to_sql` that if the table already exists, which it does in this case, then the values from the data frame should be inserted into the table. Lastly, with `index=False` we tell `to_sql` not to include the index as an additional column in the table.

## Conclusion

Now we have a SQLite database that is ready to go! You can query the *superstore.db* database file with the SQLite command line shell or other database software that supports SQLite, such as DbVisualizer. You can also run queries on the database using `read_sql` from *pandas* (see more information here). For instance, if we want to see the first five rows from *sales* we could run the code shown below in our Python shell or notebook.

```
pd.read_sql("SELECT * FROM sales LIMIT 5", db_conn)
```



```
In [61]: pd.read_sql("SELECT * FROM sales LIMIT 5", db_conn)
Out[61]:
   SalesID         OrderID        ProductID     Sales  Quantity  Discount    Profit
0        1  CA-2017-152156  FUR-BO-10001798  261.9600         2      0.00   41.9136
1        2  CA-2017-152156  FUR-CH-10000454  731.9400         3      0.00  219.5820
2        3  CA-2017-138688  OFF-LA-10000240   14.6200         2      0.00    6.8714
3        4  US-2016-108966  FUR-TA-10000577  957.5775         5      0.45 -383.0310
4        5  US-2016-108966  OFF-ST-10000760   22.3680         2      0.20    2.5164
```

When you are finished working on this database in Python you can close the connection by running `db_conn.close()` . If you want to work with this database again after closing, just reestablish the database connection

```
db_conn = sqlite3.connect("data/superstore.db")
```

and run queries using `read_sql` . No need to establish a new cursor if you are not writing data to the database.

One parting word of clarification before we close. At the start of this post, I mentioned that the structure of an Excel workbook is similar to a SQL database in that both can be comprised of multiple related tables. To that end, this exercise is intended to demonstrate how an Excel workbook can easily be turned into a SQLite database file. Nonetheless, we just scratched the surface of database design and I want to stress that a well-designed database is more than just a collection of tables. Indeed, the process of **normalization**, which aims to remove data redundancies and promote data integrity, is very important in relational database design. Be aware that the database that we created could be normalized further. If you are interested in learning more about relational databases and the rationale behind storing data across multiple tables I do recommend reading more about normalization.

Alright, that will do it for this post. Thanks for reading and please reach out if you have any questions and/or constructive feedback.

Sqlite Python Database Pandas Excel

**Written by Scott A. Adams**

108 Followers · Writer for Towards Data Science

Follow