Run SQL data queries with pandas

How-to: Run SQL data queries with pandas

Use pandas to do joins, grouping, aggregations, and analytics on datasets in Python.

By Yuli Vasiliev | *March 2021*

Python's pandas library, with its fast and flexible data structures, has become the de facto standard for data-centric Python applications, offering a rich set of built-in facilities to analyze details of structured data. Built on top of other core Python libraries, such as NumPy, SQLAlchemy, and Matplotlib, pandas leverages these libraries behind the scenes for quick and easy data manipulations, allowing you to take advantage of their functionality with less coding. For example, the `read_sql()` and `to_sql()` pandas methods use SQLAlchemy under the hood, providing a unified way to send pandas data in and out of a SQL database.

This article illustrates how you can use pandas to combine datasets, as well as how to group, aggregate, and analyze data in them. For comparison purposes, you'll also see how these same tasks can be addressed with SQL.

Creating database structures for article examples

To follow along with the examples in this article, you need to create several example tables in an Oracle database by executing the `pandas_article.sql` script that accompanies the article. Also make sure you have the pandas, SQLAlchemy, and cx_Oracle libraries installed in your Python environment. You can install them using the `pip` command:

```
pip install pandas
pip install SQLAlchemy
pip install cx_Oracle
```

Copy

For details on how to install pandas, refer to the documentation. For SQLAlchemy installation details, refer to the SQLAlchemy documentation. For details on how to install cx_Oracle, refer to the cx_Oracle Installation page. You might also want to look at the cx_Oracle Initialization page.

Loading data from Oracle Database to pandas DataFrames

After executing the `pandas_article.sql` script, you should have the orders and details database tables populated with example data. The following script connects to the database and loads the data from the orders and details tables into two separate DataFrames (in pandas, DataFrame is a key data structure designed to work with tabular data):

```
import pandas as pd
import cx_Oracle
import sqlalchemy
from sqlalchemy.exc import SQLAlchemyError
try:
   engine = sqlalchemy.create_engine("oracle+cx_oracle://usr:pswd@localhost/?serv
   orders_sql = """SELECT * FROM orders""";
   df_orders = pd.read_sql(orders_sql, engine)
   details_sql = """SELECT * FROM details""";
   df_details = pd.read_sql(details_sql, engine)
   print(df_orders)
   print(df_details)
except SQLAlchemyError as e:
   print(e)
```

In this example, you use `sqlalchemy` to create an engine to connect to an Oracle database. Using a SQLalchemy engine allows you to pass in the `arraysize` argument that will be used when `cx_Oracle.Cursor` objects are created.

The `arraysize` attribute of the `cx_Oracle.Cursor` object is used to tune the number of rows internally fetched and buffered when fetching rows from `SELECT` statements and `REF CURSOR`. By default, this attribute is set to 100, which is perfectly acceptable when you need to load a small amount of data from the database. However, when you're dealing with large amounts of data, you should increase the value of `arraysize` to reduce the number of round trips between your script and the database and, therefore, improve performance. For further details on how you can use `arraysize`, refer to the Tuning cx_Oracle documentation page.

The script prints the `df_orders` and `df_details` DataFrames loaded from the database, producing the following output:

```
      PONO      ORDATE         EMPL
0  7723510  2020-12-15  John Holland
1  5626234  2020-12-15     Tim Lewis
2  7723533  2020-12-15  John Holland
3  7823675  2020-12-16    Maya Candy
4  5626376  2020-12-16     Tim Lewis
5  5626414  2020-12-17      Dan West
6  7823787  2020-12-17    Maya Candy
7  5626491  2020-12-17      Dan West

       PONO  LINEID          ITEM        BRAND   PRICE  QUANTITY  DISCOUNT
0   7723510       1   Swim Shorts       Hurley   17.95         1         0
1   7723510       2        Jacket       Oakley  142.33         1         0
2   5626234       1         Socks         Vans   16.15         4        15
3   7723533       1         Jeans    Quiksilver   84.90         2        25
4   7723533       2         Socks  Mons Royale   10.90         2         0
5   7723533       3         Socks       Stance   12.85         2        20
6   7823675       1       T-shirt    Patagonia   35.50         3         0
7   5626376       1         Hoody       Animal   44.05         1         0
8   5626376       2  Cargo Shorts       Animal   38.60         1        12
9   5626414       1         Shirt       Volcom   78.55         2         0
10  7823787       1  Boxer Shorts     Superdry   30.45         2        18
11  7823787       2        Shorts        Barts   35.90         1         0
12  5626491       1  Cargo Shorts     Billabong   48.74         1        22
13  5626491       2       Sweater      Dickies   65.95         1         0
```

Copy

Joining DataFrames

The `DataFrame.merge()` method is designed to address this task for two DataFrames. The method allows you to explicitly specify columns in the DataFrames, on which you want to join those DataFrames. You can also specify the type of join to produce the desired result set. By default, `merge()` creates an inner join on the column that the DataFrames being joined have in common. So, you can join the `df_orders` and `df_details` DataFrames created in the previous section with the following simple call of `merge()`:

```
df_orders_details = df_orders.merge(df_details)
```

Copy

If you print the `df_orders_details` DataFrame, it should look as follows:

```
    PONO     ORDATE      EMPL              LINEID  ITEM          BRAND        PRI
0   7723510  2020-12-15  John Holland      1       Swim Shorts   Hurley       17.
1   7723510  2020-12-15  John Holland      2       Jacket        Oakley       142
2   5626234  2020-12-15     Tim Lewis      1       Socks         Vans         16.
3   7723533  2020-12-15  John Holland      1       Jeans         Quiksilver   84.
4   7723533  2020-12-15  John Holland      2       Socks         Mons Royale  10.
5   7723533  2020-12-15  John Holland      3       Socks         Stance       12.
6   7823675  2020-12-16    Maya Candy      1       T-shirt       Patagonia    35.
7   5626376  2020-12-16     Tim Lewis      1       Hoody         Animal       44.
8   5626376  2020-12-16     Tim Lewis      2       Cargo Shorts  Animal       38.
9   5626414  2020-12-17      Dan West      1       Shirt         Volcom       78.
10  7823787  2020-12-17    Maya Candy      1       Boxer Shorts  Superdry     30.
11  7823787  2020-12-17    Maya Candy      2       Shorts        Barts        35.
12  5626491  2020-12-17      Dan West      1       Cargo Shorts  Billabong    48.
13  5626491  2020-12-17      Dan West      2       Sweater       Dickies      65.
```

Copy

After joining two datasets into a single one, you may still need to modify it before you can perform analysis. In the case of `df_orders_details` being discussed here, you might need to add some new columns, calculating their values based on the values in the existing columns. Thus, you might need to add a `TOTAL` column that contains the extended item price (price multiplied by quantity and minus discount), for example:

```
df_orders_details['TOTAL'] = df_orders_details.PRICE * df_orders_details.QUANTITY
```

Copy

Copy

Since all the float columns in the `df_orders_details` DataFrame contain monetary values, you can specify two decimal places to round each float column to

```
df_orders_details = df_orders_details.round(2)
```

Copy

Some columns in the DataFrame may not be needed for the analysis you want to perform. So, you can keep only those columns that are needed. In the `df_orders_details` DataFrame, for example, if you want to group sales data (both totals and discounts) by order dates and employees, you can keep just these four columns:

```
df_sales = df_orders_details[['ORDATE','EMPL', 'TOTAL', 'OFF']]
```

Copy

If you print the DataFrame, it will look like this:

```
        ORDATE          EMPL   TOTAL    OFF
0   2020-12-15  John Holland   17.95   0.00
1   2020-12-15  John Holland  142.33   0.00
2   2020-12-15     Tim Lewis   54.91   9.69
3   2020-12-15  John Holland  127.35  42.45
4   2020-12-15  John Holland   21.80   0.00
5   2020-12-15  John Holland   20.56   5.14
6   2020-12-16    Maya Candy  106.50   0.00
7   2020-12-16     Tim Lewis   44.05   0.00
8   2020-12-16     Tim Lewis   33.97   4.63
9   2020-12-17      Dan West  157.10   0.00
10  2020-12-17    Maya Candy   49.94  10.96
11  2020-12-17    Maya Candy   35.90   0.00
12  2020-12-17      Dan West   38.02  10.72
13  2020-12-17      Dan West   65.95   0.00
```

Copy

```
CREATE VIEW sales_v AS
SELECT
  ordate,
  empl,
  price*quantity*(1-discount/100) AS total,
  price*quantity*(discount/100) AS off
FROM orders INNER JOIN details
ON orders.pono = details.pono;
```

Copy

Grouping and aggregating data

Using the `DataFrame.groupby()` method you can split a DataFrame's data into subsets (groups) that have matching values for one or more columns, and then apply an aggregate function to each group. In the following example, you group by the `ORDATE` and `EMPL` columns in the `df_sales` DataFrame and then apply the `sum()` aggregate function to the `TOTAL` and `OFF` columns within the formed groups:

```
df_date_empl = df_sales.groupby(['ORDATE','EMPL']).sum()
```

Copy

The generated DataFrame should look as shown below:

```
                              TOTAL    OFF
ORDATE      EMPL
2020-12-15 John Holland  329.99  47.59
           Tim Lewis      54.91   9.69
2020-12-16 Maya Candy    106.50   0.00
           Tim Lewis      78.02   4.63
2020-12-17 Dan West      261.07  10.72
           Maya Candy     85.84  10.96
```

Copy

One problem here is that the aggregate function you apply to the `groupby` object is applied to each numeric column of the DataFrame. But what if you need to apply multiple aggregate functions to multiple `groupby`

```
df_aggs = df_sales.groupby(['ORDATE','EMPL']).agg({'TOTAL': ['sum', 'mean'], 'OFF
```

Copy

The above example illustrates how you can select a certain column for aggregation and perform different aggregations per column. If you print `df_aggs`, it will look as follows:

```
                           TOTAL           OFF
                             sum    mean    max
ORDATE     EMPL
2020-12-15 John Holland   329.99   66.00  42.45
           Tim Lewis       54.91   54.91   9.69
2020-12-16 Maya Candy     106.50  106.50   0.00
           Tim Lewis       78.02   39.01   4.63
2020-12-17 Dan West       261.07   87.02  10.72
           Maya Candy      85.84   42.92  10.96
```

Copy

You might want to flatten a hierarchical index in columns. This can be done as follows:

```
df_aggs.columns = df_aggs.columns.map('_'.join).str.strip()
```

Copy

This will change the column names as shown below:

```
TOTAL_sum  TOTAL_mean  OFF_max
ORDATE     EMPL
...
```

Copy

To generate the same result set with a query to the article sample database, you could issue the following `SELECT` statement against the `sales_v` view that you should have created previously:

```
    empl,
    ROUND(SUM(total),2) TOTAL_sum,
    ROUND(AVG(total),2) TOTAL_mean,
    ROUND(MAX(off),2) OFF_max
FROM
    sales_v
GROUP BY
    ordate, empl
ORDER BY
ordate;
```

Copy

Analytical processing within groups of data

In practice, you may not always need to view data in summarized format, aggregating a group of rows into a single resulting row as illustrated in the previous example. In contrast, you may need to do some analytical processing within a group of rows so the number of rows in the group remains the same. For example, if you want to compare the salary of each employee in a department with the average salary of the employees in this department, this processing does not imply any reduction in the number of rows in the dataset—the number of rows must match the number of employees, both before and after processing.

Let's illustrate this analytical processing with a second, more complex example. Imagine you want to analyze stock price data for a list of tickers over a certain period of time. To start, you want to weed out the tickers whose prices dropped below 1% of the previous day's price over the period. To accomplish this, you need to group data by ticker symbol, ordering the rows by date in each group. Then you can iterate over the rows in a group, comparing the stock price in the current row with the price in the previous row. If the price in a current row is less than the price in the previous row by more than 1%, then the entire group of rows must be excluded from the result set. This section describes how you could implement this filtering.

The example uses stock data obtained via the yfinance library, a Python wrapper for the Yahoo Finance API, which you can install with the `pip` command, as follows:

```
pip install yfinance
```

Copy

In the following script, you get stock data for several popular stocks for a five-day period:

```
import pandas as pd
import yfinance as yf
```

```
    hist = tkr.history(period='5d')
    hist['Symbol']=ticker
    stocks = stocks.append(hist[['Symbol', 'Close']].rename(columns={'Close': 'Pric
```

Copy

yfinance returns a requested dataset as a pandas DataFrame with the `Date` column as the index. Assuming you are targeting the closing prices only, you keep only the `Symbol` and `Close` columns, having renamed the latter to `Price` for comprehension. As a result, the data in the DataFrame might look like this:

```
           Symbol       Price
Date
2020-12-18   AAPL   126.660004
2020-12-21   AAPL   128.229996
2020-12-22   AAPL   131.880005
2020-12-23   AAPL   130.960007
2020-12-24   AAPL   131.970001
2020-12-18   TSLA   695.000000
2020-12-21   TSLA   649.859985
2020-12-22   TSLA   640.340027
2020-12-23   TSLA   645.979980
2020-12-24   TSLA   661.770020
2020-12-18     FB   276.399994
2020-12-21     FB   272.790009
2020-12-22     FB   267.089996
2020-12-23     FB   268.109985
2020-12-24     FB   267.399994
2020-12-18   ORCL    65.059998
2020-12-21   ORCL    64.480003
2020-12-22   ORCL    65.150002
2020-12-23   ORCL    65.300003
2020-12-24   ORCL    64.959999
2020-12-18   AMZN  3201.649902
2020-12-21   AMZN  3206.179932
2020-12-22   AMZN  3206.520020
2020-12-23   AMZN  3185.270020
2020-12-24   AMZN  3172.689941
```

Copy

From the above row set, you need to select the rows related to only those symbols whose prices did not drop below 1% of the previous day's price. For this, you need a mechanism that will allow you to compare the `Price` value of a row with the `Price` value of the previous row within a symbol group. The following line of code

```
stocks['Prev'] = stocks.groupby(['Symbol'])['Price'].shift(1)
```

Copy

The updated row set should look as follows:

```
           Symbol        Price         Prev
Date
2020-12-18   AAPL   126.660004          NaN
2020-12-21   AAPL   128.229996   126.660004
2020-12-22   AAPL   131.880005   128.229996
2020-12-23   AAPL   130.960007   131.880005
2020-12-24   AAPL   131.970001   130.960007
2020-12-18   TSLA   695.000000          NaN
2020-12-21   TSLA   649.859985   695.000000
2020-12-22   TSLA   640.340027   649.859985
2020-12-23   TSLA   645.979980   640.340027
2020-12-24   TSLA   661.770020   645.979980
2020-12-18     FB   276.399994          NaN
2020-12-21     FB   272.790009   276.399994
2020-12-22     FB   267.089996   272.790009
2020-12-23     FB   268.109985   267.089996
2020-12-24     FB   267.399994   268.109985
2020-12-18   ORCL    65.059998          NaN
2020-12-21   ORCL    64.480003    65.059998
2020-12-22   ORCL    65.150002    64.480003
2020-12-23   ORCL    65.300003    65.150002
2020-12-24   ORCL    64.959999    65.300003
2020-12-18   AMZN  3201.649902          NaN
2020-12-21   AMZN  3206.179932  3201.649902
2020-12-22   AMZN  3206.520020  3206.179932
2020-12-23   AMZN  3185.270020  3206.520020
2020-12-24   AMZN  3172.689941  3185.270020
```

Copy

The `Prev` results for the first day of the observation period are `NaN` because this example does not track what happened before the five-day range.

Now you can find those rows where the ratio of the price to the previous price is less than 99%, for example:

```
stocks_to_exclude = stocks[stocks['Price']/stocks['Prev'] < .99]
```

```
Date
2020-12-21   TSLA  649.859985  695.000000
2020-12-22   TSLA  640.340027  649.859985
2020-12-21     FB  272.790009  276.399994
2020-12-22     FB  267.089996  272.790009
```

Copy

You can extract the symbols presented in the above rows as follows:

```
exclude_list = list(set(stocks_to_exclude['Symbol'].tolist()))
```

Copy

Here you extract the values of the `Symbol` column in the `stocks_to_exclude` DataFrame, converting those values to a list. To exclude duplicates from this list, you convert it into a set and then back to a list (one of the most popular ways to remove duplicates from a list).

```
['TSLA', 'FB']
```

Copy

Next you need to exclude those rows from the `stocks` DataFrame that includes the above names in the `Symbol` field. This can be implemented as the following one-liner:

```
stocks_filtered = stocks[~stocks['Symbol'].isin(exclude_list)][['Symbol', 'Price'
```

Copy

You pass in the `exclude_list` that you created previously to the `stocks['Symbol'].isin()` function to get a `Series` of booleans indicating if each value in the `stocks['Symbol'] Series` is in `exclude_list`. You put a tilde sign in front of `stocks['Symbol']` to invert the boolean Series returned by `isin()`. You pass in this inverted boolean `Series` to the `[]` operator of the `stocks` DataFrame to return all the rows that do not contain in the `Symbol` column symbols found in the exclude_list. So, the resulting DataFrame should look as follows:

```
2020-12-21   AAPL   128.229996
2020-12-22   AAPL   131.880005
2020-12-23   AAPL   130.960007
2020-12-24   AAPL   131.970001
2020-12-18   ORCL    65.059998
2020-12-21   ORCL    64.480003
2020-12-22   ORCL    65.150002
2020-12-23   ORCL    65.300003
2020-12-24   ORCL    64.959999
2020-12-18   AMZN  3201.649902
2020-12-21   AMZN  3206.179932
2020-12-22   AMZN  3206.520020
2020-12-23   AMZN  3185.270020
2020-12-24   AMZN  3172.689941
```

Copy

With the help of analytical SQL, you can get the same result set with a single query to the database. Before you can do this, however, you need to save the unfiltered row set to the article database, which should contain the stocks table for storing this data (refer back to the `pandas_article.sql` script that you should have run at the very beginning).

To conform to the structure of the stocks database table, you need to modify the `stocks` DataFrame that contains the unfiltered data of this example. This can be done with the following lines of code:

```
stocks_to_db = stocks[['Symbol', 'Price']].reset_index().rename(columns={'Date':
stocks_to_db = stocks_to_db.astype({'Dt': str})
```

Copy

In the first line, you specify the columns to include in the result set: `Symbol` and `Price`. By resetting the index, you add `Date` to this column list. To conform to the name of this column in the stocks database table, you rename it to `Dt`. The `round(2)` function rounds the values in the `Price` column to two decimal places. In the second line, you cast the `Dt` column to the `str` type, because pandas sets it to datetime by default.

Finally, you need to convert the `stocks_to_db` DataFrame to a structure that is passable to a method that can do a bulk insert operation. In the following line of code, you convert the `stocks_to_db` DataFrame to a list of tuples:

```
data = list(stocks_to_db.itertuples(index=False, name=None))
```

The following script uses the above list of tuples to upload the data it contains to the database. Storing the data you work with can be useful when you're going to reuse it.

```
import cx_Oracle
try:
  conn = cx_Oracle.connect("usr", "pswd", "localhost/orcl")
  cursor = conn.cursor()
  #defining the query
  query_add_stocks = """INSERT INTO stocks (dt, symbol, price)
                        VALUES (TO_DATE(:1, 'YYYY-MM-DD'), :2, :3)"""
  #inserting the stock rows
  cursor.executemany(query_add_stocks, data)
  conn.commit()
except cx_Oracle.DatabaseError as exc:
  err = exc.args
  print("Oracle-Error-Code:", err.code)
  print("Oracle-Error-Message:", err.message)
finally:
  cursor.close()
  conn.close()
```

Copy

In this script, you connect to the database and obtain a cursor object to interact with it. You use the `cursor.executemany()` method that inserts all the rows from the data list of tuples into the database in a single round trip.

After the successful execution of the above script, you can issue queries against the stocks table. To get the row set you had in the `stocks_filtered` DataFrame, you can issue the following query:

```
SELECT s.* FROM stocks s
LEFT JOIN
(SELECT DISTINCT(symbol) FROM
  (SELECT price/LAG(price) OVER (PARTITION BY symbol ORDER BY dt) AS dif, symbol
ON a.symbol = s.symbol WHERE a.symbol IS NULL;
```

Copy

Conclusion

reshaping original datasets as needed.

Dig deeper

- Learn more about how to install pandas.
- Learn more about how to install SQLAlchemy.
- Learn more about how to install cx_Oracle.
- Read the cx_Oracle Initialization guide.
- Get the sample dataset for this article.

Illustration: Wes Rowell

Yuli Vasiliev is a programmer, freelance writer, and consultant. He is the author of *Natural Language Processing with Python and spaCy*.