Converting Excel spreadsheets to SQLite with Python

marcus

May 22, 2024

Contents

1 README

This is a short example to show how to

1. Write SQLite data to CSV files

2. Import CSV files into Excel workbook

3. Read Excel data into a Python dataframe

4. Create an SQLite database for the data

5. Insert the data into the SQLite database

This was written for neo-Pythonistas who already know and understand SQLite database design and manipulation.

Source: "Turn Your Excel Workbook Into a SQLite Database" by S.A. Adams (May 18, 2020). URL: towardsdatascience.com.

2 Write SQLite data to a CSV file

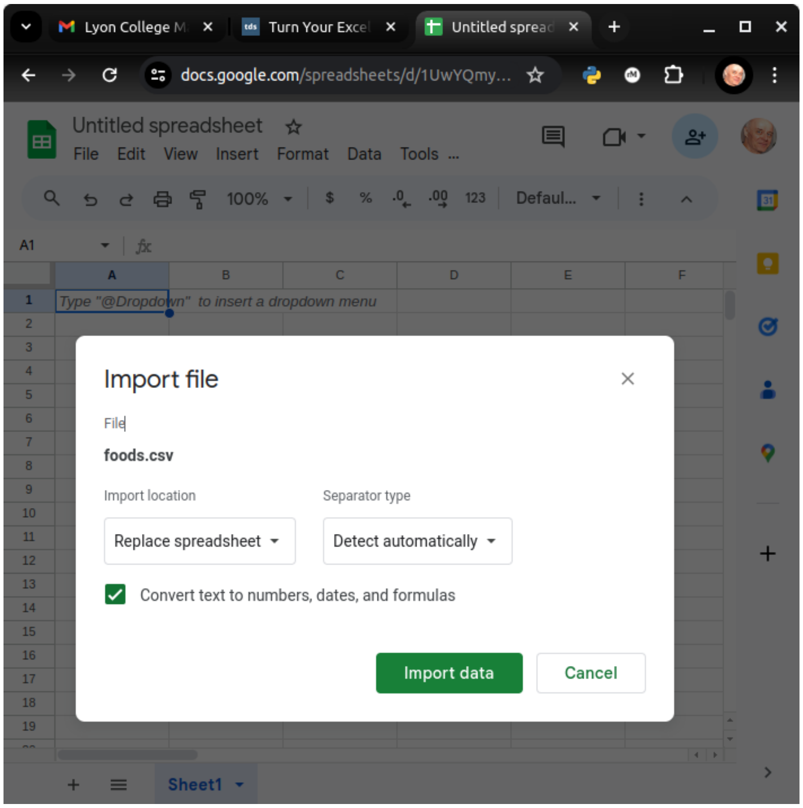
- Instead of the article data, I am using a very simple test file (downloadable tinyurl.com/test-data-csv):

```
id,value
1,"Ms. Jane Robinson"
2,"Mr. Carl Robinson"
3,"Mr. Edward Jones"
```

3 Turn the CSV file into an Excel file

- You either need Microsoft Excel for this, or a free clone like LibreOffice spreadsheet, or Google Docs.

- I'm using Google Docs to `import` the CSV files:



Import file

File

foods.csv

Import location

Replace spreadsheet

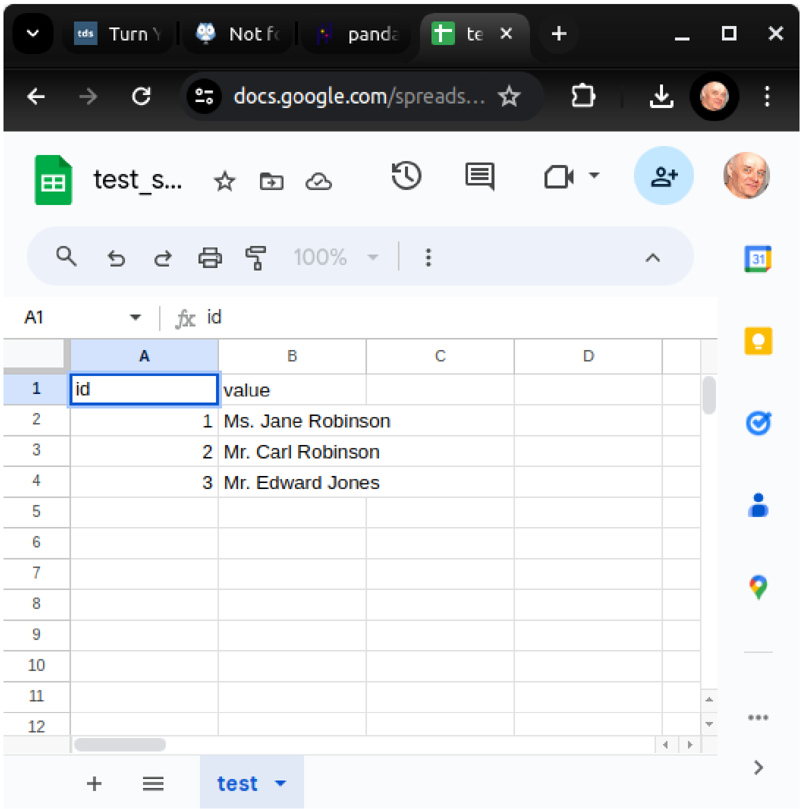
Separator type

Detect automatically

☑ Convert text to numbers, dates, and formulas

Import data
Cancel
Lyon College M
Turn Your Excel
Untitled spread
docs.google.com/spreadsheets/d/1UwYQmy...
Untitled spreadsheet
File Edit View Insert Format Data Tools
100%
Defaul...
A1
Type "@Dropdown" to insert a dropdown menu
Sheet1

- The result:



	A	B
1	id	value
2	1	Ms. Jane Robinson
3	2	Mr. Carl Robinson
4	3	Mr. Edward Jones

- This is the file that we'll read into a Python `Data.Frame`. You can find the online file here for download: tinyurl.com/excel-to-sqlite-csv

4 Read Excel data into a Python `DataFrame`

- Python is an all-purpose high-level programming language used much in data science in machine learning but also useful for general scripting and automating of tasks.
- For data science, the `pandas` package is especially useful: as you can read in the online documentation, `pandas` provides data analysis tools to Python.
- If you do this in an interactive DataCamp DataLab or Google Colab notebook, `pandas` will already be installed and you only have to load it[1].

[1] You do not need a fancy setup with the `conda` platform if you use an interactive

- To use `pandas`, you have to `import` the library:

  ```
  import pandas as pd
  ```

- Now, you have access to `pandas` functions, e.g. `pd.read_excel`:

  ```
  help(pd.read_excel)
  ```

- Here’s the top of the `help` output (indentation changed):

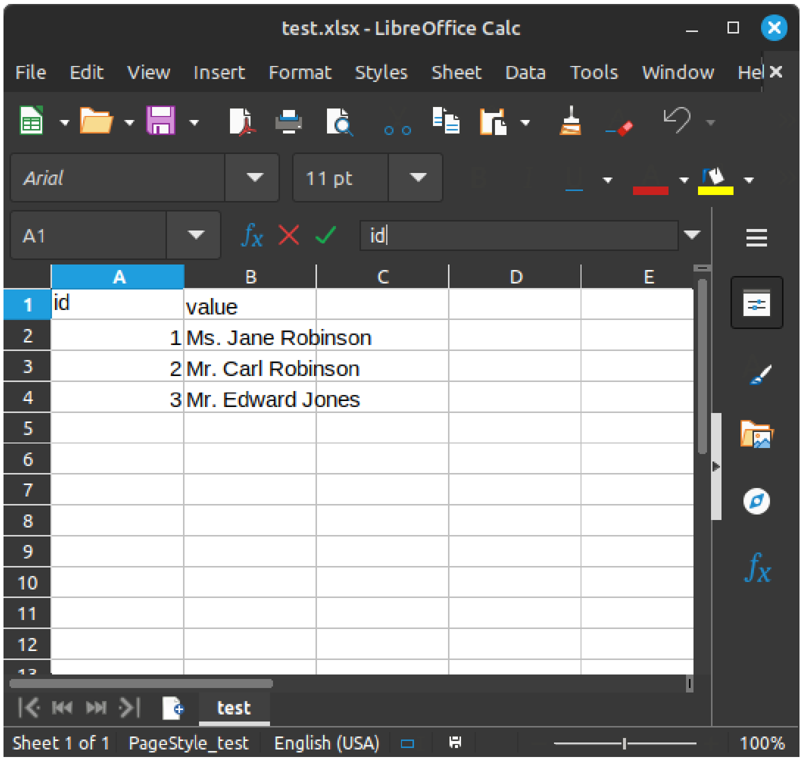
  ```
  read_excel(io,
             sheet_name: ’str | [...]
             header: ’int | Sequence[int] | None’ = 0, ...]’

      Read an Excel file into a ‘‘pandas‘‘ ‘‘DataFrame‘‘.

      Supports ‘xls‘, ‘xlsx‘, ‘xlsm‘, ‘xlsb‘, ‘odf‘, ‘ods‘ and ‘odt‘
      file extensions. Read from a local filesystem or URL. Supports
      an option to read a single sheet or a list of sheets.

      [...]

      See Also
      --------
      DataFrame.to_excel : Write DataFrame to an Excel file.
      DataFrame.to_csv : Write DataFrame to comma-separated values file
      read_csv : Read a comma-separated values (csv) file into DataFrame.
      read_fwf : Read a table of fixed-width formatted lines into DataFrame.
  ```

- You can find out much more about `read_excel` in the online documentation. As you can see in the `help`, the function only has one mandatory argument `io`, which can be a URL string or an Excel file name (in quotes).

- The `header` parameter is 0 by default - we’re OK because we got one.

- URL import, especially from Google Docs, does not always work: to be on the safe side, I’ve downloaded the Excel file as `test.xlsx`:

(’Jupyter’) notebook installation in the cloud. If you’re using Emacs (which is what I do), you’re also set (locally). What I’ve done is write all of this as a literate program in Emacs, which I will then render as an IPython notebook (`excel_to_sqlite.ipynb`), upload to DataLab and share with you.



id

	A	B
1	id	value
2	1	Ms. Jane Robinson
3	2	Mr. Carl Robinson
4	3	Mr. Edward Jones

- We import the data into a `Data.Frame` named `df`:

```
df = pd.read_excel('test.xlsx')
print(df.head())
```

```
   id              value
0   1  Ms. Jane Robinson
1   2  Mr. Carl Robinson
2   3   Mr. Edward Jones
```

- You see that there's an extra column for the row index starting at 0. The `pandas` function `info` provides overall information:

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
```

```
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      3 non-null      int64
 1   value   3 non-null      object
dtypes: int64(1), object(1)
memory usage: 176.0+ bytes
None
```

5 Create SQLite database and put the data into it

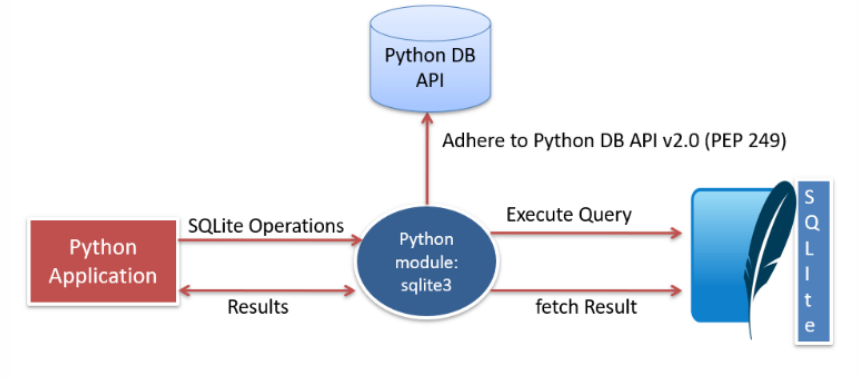


Figure 1: Source: pynative.com/python-sqlite/

- We're now going to create a `test.db` SQLite database using Python's `sqlite3` package, which needs to be imported (or installed):

  ```
  import sqlite3
  ```

- As you can read in the documentation, `sqlite3` is a database interface for SQLite databases: it allows you to submit SQLite commands from within a Python script. There is also a tutorial.

- The image illustrates how the Python modulde `sqlite3` works:

  1. You run SQLite operations (like `SELECT`) in Python and results are returned to the Python console.
  2. The `sqlite3` module executes queries on the SQLite database, and fetches results from the SQLite database.

3. To establish data transfer between database and Python script, there is an Application Programming Interface (API), PEP 249.

- The steps to hitching SQLite to Python are as follows:
  1. With sqlite3.connect, initiate a new SQLite database connection object db_conn, which creates an (empty) database test.db.
  2. Run a cursor object on the connection. This object lets us execute SQLite data definition commands like CREATE TABLE.
  3. Run the pandas function to_sql on a DataFrame to INSERT data into an SQLite table.
  4. To execute SQLite queries on a given database, run SELECT commands on the tables using the pandas function read_sql.

6 Initiate a database connection creating an empty database

- Remove the test.db database if it already exists:

```
import os
os.system("rm ../data/test.db")
```

- Creating a connection object also creates an (empty) database:

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

- Type of object:

```
print(type(db_conn))
```

```
<class 'sqlite3.Connection'>
```

- Check the empty database (os.system executes OS shell commands):

```
os.system("ls -l ../data/test.db")
```

```
-rw-r--r-- 1 marcus marcus 0 May 22 14:19 ../data/test.db
```

7 Run data definition commands on the database to create tables

- Data definition means that we need to devise a schema.
- We want a very simple database schema:

  ```
  CREATE TABLE test (id INTEGER PRIMARY KEY,
                     value TEXT);
  ```

- The DataFrame objects where we stored the data, are already aligned with this database design (apart from the bridge table foods_episodes):

  ```
  print(df.columns)

  Index(['id', 'value'], dtype='object')
  ```

8 SQLite database reference cursor

- This is the database design that we're now going to build using the Cursor object db_conn.cursor - a reference pointing at the database:

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

  <class 'sqlite3.Cursor'>
  ```

- You can get help on this object directly, or check the documentation[2]

  ```
  help(db_conn.cursor())
  ```

- Now create the table test using the reference to test.db:

[2] The cursor may appear like a pointless abstraction to you - why not just use the connection object? The reason is encapsulation of SQLite commands - the connection manages the connection to the database, while the cursor contains methods to execute SQLite commands. The cursor also maintains the state of the current query, which is critical for fetching data in chunks and adds efficiency.

```
c.execute(
    """
    CREATE TABLE
       IF NOT EXISTS
       test (
       id INTEGER PRIMARY KEY,
       value TEXT
       );
    """
)
```

- Check that the table was created:

```
tab = c.execute("SELECT name FROM sqlite_master")
print(tab.fetchone())
os.system("ls -l ../data/test.db")
```

```
('test',)
-rw-r--r-- 1 marcus marcus 8192 May 22 14:19 ../data/test.db
```

- The query returns a tuple containing the table's name `test` - still empty except for the table definition.

9 Insert data from the `DataFrame` into database tables

- This command transfers the content of `df` to the `test` table in our database.

```
df.to_sql('test',  # target table
          db_conn, # database connection
          if_exists='append', # append data if table exists
          index=False)  # do not add DataFrame index as a table column
```

10 Run queries on the database tables

- To run queries on the data, we use `pandas` function `read_sql`. The first argument is the command, the second the database connection:

```
query = pd.read_sql("SELECT * FROM test", db_conn)
print(query)
```

```
   id              value
0   1  Ms. Jane Robinson
1   2  Mr. Carl Robinson
2   3   Mr. Edward Jones
```

- The first column is not a table column but the index column of the output `DataFrame`:

```
print(query.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      3 non-null      int64
 1   value   3 non-null      object
dtypes: int64(1), object(1)
memory usage: 176.0+ bytes
None
```

- The `read_sql` function is a wrapper around two other functions from the `SQLAlchemy` toolkit - if you want to get more deeply into writing Python scripts for database access, check out the documentation.