SQL in Python

Chapter 9

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Python in R

The sqlite3 Module

The pandas Library

The sqlalchemy Library

Python in R

The reticulate Package

The R package **`reticulate`** contains functions and tools to allow R to interface with Python.

```
library(reticulate)
```

In particular, after loading `reticulate`, we can create `python` code chunks, similar to R and SQL code chunks.

The syntax ```` ```{python chunk_name} ```` is used to define Python code chunks.

Setting Up a Virtual Environment

It is common practice in Python to create and use virtual environments to ensure encapsulation of packages and avoid version conflicts.

The commands below create a virtual environment called `stats167_venv`, install necessary packages, and activate/use the virtual environment.

```
library(reticulate)
# If needed, install Python:
# install_python(version = "3.10.8")
# Create a new virtual environment
virtualenv_create("stats167_venv")
# Install packages
virtualenv_install("stats167_venv",
   packages = c("numpy", "pandas", "sqlalchemy")
)
# Use virtual environment
use_virtualenv("stats167_venv")
```

Note: Official documentation for `reticulate` no longer recommends self-managing virtual environments, instead recommending the `py_require()` function.

From Python to R

To verify we have correctly installed and loaded `reticulate`, we can create a Python code chunk and run some small code.

```
import pandas as pd
births = pd.read_csv("births.csv")
print(births.shape)
```

```
(1998, 21)
```

Any objects we create in Python can be accessed within R code chunks by using the `py` object in R. The `$` operator is used to extract specific objects from `py`.

```
dim(py$births)
```

```
[1] 1998   21
```

There and Back Again

Going the other way around, the `r` object in Python can be used to access R objects within Python code chunks. The period (`.`) operator is used to extract specific objects from `r`.

```
print(r.trees.head())
```

```
   Girth  Height  Volume
0    8.3    70.0    10.3
1    8.6    65.0    10.3
2    8.8    63.0    10.2
3   10.5    72.0    16.4
4   10.7    81.0    18.8
```

```
print(r.trees.shape)
```

```
(31, 3)
```

The `r_to_py()` and `py_to_r()` functions can also be used to translate specific R objects to Python and vice versa.

Further Resources

We have covered the basic functions using the `reticulate` framework, mostly to facilitate Python code chunks in R Markdown.

There are other ways to use Python with R beyond what we will cover, including running Python scripts in R and creating an interactive Python console within an R console.

Further information can be found in the official documentation: https://rstudio.github.io/reticulate/

The sqlite3 Module

The sqlite3 Module

Similar to the `DBI` package in R, the **`sqlite3`** module in Python provides a SQLite database interface for Python to connect to and interact with databases.

The `sqlite3` module is pre-installed in the standard Python library, so there is no separate installation required. To access the module, we just need to import the module.

```
import sqlite3
```

Note that the `sqlite3` module is only for connecting to and interacting with SQLite databases.

PEP 249

Even though each dialect of SQL has a different database API (application programming interface) module, each module follows to the Python DB-API 2.0 specifications (called PEP 249) so that they all share the same basic structure and syntax for connecting and querying to databases.

Some common DB-API modules:

- `pymysql` or `mysql-connection-python` for MariaDB/MySQL
- `psycopg` for PostgreSQL
- `pyodbc` for (Microsoft) ODBC (SQL Server)
- `python-oracledb` for Oracle

Since all these modules follow PEP 249, understanding the tools in `sqlite3` will translate to other SQL dialects when you need them.

The sqlite3.connect() Function

The first step to accessing data from a relational database is to make a connection to it. The **`sqlite3.connect()`** function establishes a connection to a SQLite database.

```
con = sqlite3.connect("TYSQL.sqlite")
```

Once a `Connection` object has been made, we can now access our database using the connection.

Cursors

Before we can use the connection to query from the database, we need to create a cursor object.

In Python, a **cursor** is an object used to send SQL commands to and retrieve (or **fetch**) results from a database connection.

A cursor helps manage local memory resources by controlling the flow and timing of data fetching. For large datasets, fetching results in smaller batches (rather than all at once) uses less memory and makes the process more efficient.

<An animated gif illustrating a cursor.>

Side Note: In R, the `DBI::dbGetQuery()` function automatically manages the cursor, so there is no need to explicitly create one.

Note: A cursor in SQL is a similar but different object. A SQL cursor also controls the flow and timing of query execution and result fetching, but it is managed by the DBMS (i.e., the server) rather than the local interface (i.e., the client).

The Connection.cursor() Method

The **`Connection.cursor()`** method creates a cursor object that uses the `Connection` to query and fetch from the database.

```
cur = con.cursor()
```

The `Cursor` object has several execute and fetch methods to interact with the database:

- `execute()`
- `executemany()`
- `executescript()`
- `fetchone()`
- `fetchmany()`
- `fetchall()`

The Cursor.close() and Connection.close() Methods

As we saw in R, a database connection needs to be closed once it is no longer needed, and the same as true for cursors.

Closing cursors and connections can be done with the `Cursor.close()` and `Connection.close()` methods.

```
cur.close()
con.close()
```

However, this is not the recommended way to close cursors and connections.

Use a Context Manager

Rather than manually closing cursors and connections, it is strongly recommended to use the `Connection` as a context manager using the **`with`** statement.

```
with sqlite3.connect("TYSQL.sqlite") as con:
    cur = con.cursor()
    # Additional commands using cur or con go here
```

Using a context manager, there is no need to close the cursor or connection, as the context manager automatically closes both when the `with` block is finished executing.

The Cursor.execute() Method

The **`Cursor.execute()`** method uses the cursor to execute a single SQL statement.

The method inputs a SQL query as a character string. Multi-line SQL statements can be written using triple quotes (""" or ''').

```
with sqlite3.connect("TYSQL.sqlite") as con:
    cur = con.cursor()
    sql_query = """
    SELECT cust_name, cust_state, cust_email
    FROM Customers;
    """
    cur.execute(sql_query)
```

```
<sqlite3.Cursor object at 0x1311003c0>
```

The command `cur.execute(sql_query)` executes the query but the results are not immediately fetched.

Execute and Fetch All

The **cur.fetchall()** command will fetch all rows from the query result set and return them as a list of tuples.

```
with sqlite3.connect("TYSQL.sqlite") as con:
    cur = con.cursor()
    sql_query = """
    SELECT cust_name, cust_state, cust_email
    FROM Customers;
    """
    cur.execute(sql_query)
    cust_rows = cur.fetchall()
    for row in cust_rows:
        print(row)
```

```
<sqlite3.Cursor object at 0x131100540>
('Village Toys', 'MI', 'sales@villagetoys.com')
('Kids Place', 'OH', None)
('Fun4All', 'IN', 'jjones@fun4all.com')
('Fun4All', 'AZ', 'dstephens@fun4all.com')
('The Toy Store', 'IL', 'kim@thetoystore.com')
('Toy Land', 'NY', None)
```

Fetch One and Fetch Many

If the query result set is large, the **fetchone()** and **fetchmany()** methods allow for smaller numbers of rows to be fetched at a time.

```
with sqlite3.connect("TYSQL.sqlite") as con:
    cur = con.cursor()
    sql_query = """
    SELECT cust_name, cust_state, cust_email
    FROM Customers;
    """
    cur.execute(sql_query)
    first_cust = cur.fetchone() # First row
    next_few = cur.fetchmany(3) # Next 3 rows
    last_few = cur.fetchall()   # Remaining rows
    for row in last_few:
        print(row)
```

```
<sqlite3.Cursor object at 0x1311006c0>
('The Toy Store', 'IL', 'kim@thetoystore.com')
('Toy Land', 'NY', None)
```

Notice that once results from the cursor are fetched, the subsequent calls will not re-fetch the same rows (i.e., the cursor has moved).

The pandas Library

Pandas DataFrames

As we saw in the previous section, the cursor fetch methods return lists of tuples that represent the row(s) of the query result set. A list of tuples may not be a convenient way to store or access data for data analysis.

To return a more familiar data structure, we can turn the list into a `pandas DataFrame` object by using the **`pandas.DataFrame()`** constructor function.

Example: Pandas DataFrames

```
with sqlite3.connect("TYSQL.sqlite") as con:
    cur = con.cursor()
    sql_query = """
    SELECT cust_name, cust_state, cust_email
    FROM Customers;
    """
    cur.execute(sql_query)
    cust_rows = cur.fetchall()
    cust_df = pd.DataFrame(cust_rows)
    print(cust_df)
```

```
<sqlite3.Cursor object at 0x131100840>
                0   1                          2
0   Village Toys  MI  sales@villagetoys.com
1     Kids Place  OH                       None
2        Fun4All  IN     jjones@fun4all.com
3        Fun4All  AZ  dstephens@fun4all.com
4  The Toy Store  IL    kim@thetoystore.com
5       Toy Land  NY                       None
```

The Cursor.description Attribute

Notice that the resulting DataFrame from the previous example does not have meaningful column names. We want a way to add column names to the results.

The **`Cursor.description`** attribute contains the column names of the last query executed by the cursor.

For historical (backward compatability) reasons, the description attribute returns a 7-tuple for each column, but the last six items of each tuple are `None`.

Thus, to access the column names, we need to extract the first (index 0) item from each entry in the description attribute.

Example: Adding Column Names

```
with sqlite3.connect("TYSQL.sqlite") as con:
    cur = con.cursor()
    sql_query = """
    SELECT cust_name, cust_state, cust_email
    FROM Customers;
    """
    cur.execute(sql_query)
    cust_rows = cur.fetchall()
    col_names = [col[0] for col in cur.description]
    cust_df = pd.DataFrame(cust_rows, columns=col_names)
    print(cust_df)
```

```
<sqlite3.Cursor object at 0x1311008c0>
       cust_name cust_state             cust_email
0   Village Toys         MI  sales@villagetoys.com
1     Kids Place         OH                   None
2        Fun4All         IN     jjones@fun4all.com
3        Fun4All         AZ  dstephens@fun4all.com
4  The Toy Store         IL    kim@thetoystore.com
5       Toy Land         NY                   None
```

The pandas.read_sql_query() Function

For a simpler approach, the **pandas.read_sql_query()** function executes a SQL query, fetches all rows from the result set, and returns the result set as a DataFrame, all in a single function call.

```
with sqlite3.connect("TYSQL.sqlite") as con:
    sql_query = """
    SELECT cust_name, cust_state, cust_email
    FROM Customers;
    """
    cust_df = pd.read_sql_query(sql_query, con)
    print(cust_df)
```

```
        cust_name cust_state             cust_email
0    Village Toys          MI  sales@villagetoys.com
1      Kids Place          OH                   None
2         Fun4All          IN     jjones@fun4all.com
3         Fun4All          AZ  dstephens@fun4all.com
4  The Toy Store          IL    kim@thetoystore.com
5        Toy Land          NY                   None
```

Comparing sqlite3 to pandas.read_sql_query()

Internally, the `pd.read_sql_query(sql_query, con)` command is intuitively a wrapper for the following steps:

```
# 1. Create a cursor object
cur = con.cursor()
# 2. Execute the SQL query
cur.execute(sql_query)
# 3. Fetch all rows from the result set
rows = cur.fetchall()
# 4. Get column names from cursor description
columns = [col[0] for col in cur.description]
# 5. Create a DataFrame
df = pd.DataFrame(rows, columns=columns)
# 6. Close the cursor
cur.close()
```

Notice that the `pandas.read_sql_query()` function manages the creation and closing of the cursor automatically, but it still requires an existing connection to the database (which it does not close).

Why Not Always Use pandas.read_sql_query()?

For many queries, calling the `pandas.read_sql_query()` function simplifies the entire process of sending queries to and fetching data from the database into a single step.

Question: What scenarios might we not want to use `pandas.read_sql_query()`? When is it more beneficial to use an explicit cursor, execute, and fetch from `sqlite3` (or other DB-API module)?

Limitations of pandas.read_sql_query()

As is often the case, understanding the purpose and design of a function highlights both its advantages and its limitations.

- **Only for SQL queries**: The `pandas.read_sql_query()` function is only for SQL queries, i.e., any `SELECT` (or `SELECT`-type) statement that returns a query result set. Other SQL statements (e.g., `CREATE`, `INSERT`, `UPDATE`, or `DELETE`) need a more general function, such as `Cursor.execute()`.
- **No cursor control**: The `pandas.read_sql_query()` function automatically manages the cursor. For large datasets, it can be better to have manual control of the cursor to optimize the flow and timing of data fetching.
- **Fetches all data at once**: The `pandas.read_sql_query()` function fetches all results at once and stores it in memory, which can be inefficient or infeasible for large query results. Using `Cursor.fetchone()` or `Cursor.fetchmany()` allows for better resource management.

The pd.read_sql() and pd.read_sql_table() Functions

Other than `pandas.read_sql_query()`, there are two other SQL functions in `pandas`.

- The **`pandas.read_sql_table(table_name, con)`** function inputs the name of a table (`table_name`) in the database and a SQLAlchemy engine (`con`) and outputs a DataFrame of the table.

  This function is similar to `DBI::dbReadTable()` in R.

  Note: The `con` argument *must* be a SQLAlchemy engine, which acts as a connection to the database. A `Connection` object from `sqlite3.connect()` or other DB-APIs are *not supported* by `read_sql_table()`.

- The **`pandas.read_sql()`** function is a wrapper for `pandas.read_sql_query()` and `pandas.read_sql_table()`. Depending on the inputs, `pandas.read_sql()` will delegate the inferred function.

  Note: The `con` argument in `pandas.read_sql_query()` can be either a DB-API connection (like `sqlite3`) or a SQLAlchemy engine.

The sqlalchemy Library

SQLAlchemy Colab Notebook

Link to Google Colab notebook:

https://colab.research.google.com/drive/1eiMwE33karjAYmx1RZMYMUnlHvciawcH?usp=sharing