**How to Connect to SQL Databases from Python Using SQLAlchemy and Pandas**

Extract SQL tables, insert, update, and delete rows in SQL databases through SQLAlchemy



Photo by [Pascal Müller](https://unsplash.com/@millerthachiller?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank) (Modify by Author)

In a data science project, we often need to interact with **Relational databases**, such as, extracting tables, inserting, updating, and deleting rows in SQL tables. To accomplish these tasks, Python has one such library, called **SQLAlchemy**. It supports popular SQL databases, such as **PostgreSQL**, **MySQL**, **SQLite**, **Oracle**, **Microsoft SQL Server**, and others. Even better, it has built-in functionalities, which can be integrated with **Pandas**. Together, SQLAlchemy and Pandas are a perfect match to handle data management.

**Install Libraries**

Besides SQLAlchemy and pandas, we would also need to install a SQL database adapter to implement **Python Database API**. For example, we need to install “psycopg2” or “pg8000” for PostgreSQL, “mysql-connector-python” or “oursql” for MySQL, “cx-Oracle” for Oracle SQL Database, “pyodbc” or “pymssql” for Microsoft SQL Server and others. In this article, I will discuss how to integrate PostgreSQL with Python, therefore, let’s install “**psycopg2**”.

Open the anaconda prompt or command prompt and type the following commands.

pip install SQLAlchemy  
pip install pandas   
pip install psycopg2

**Import Libraries**

import sqlalchemy  
import pandas as pd

**Create Connection to the Database**

First of all, let’s create a connection with the PostgreSQL database using “**create\_engine()**” function based on a **URL**. A URL usually consists of dialect, driver, username, password, hostname, database name as well as optional arguments for additional configuration. The typical form of a database URL looks like “**dialect+driver://username:password@host:port/database**”. For example, “mssql+pyodbc://username:password@host:port/database” for Microsoft SQL Server, “mysql+mysqlconnector://username:password@host:port/database” for MySQL and “postgresql+psycopg2://username:password@host:port/database” for PostgreSQL.

url = '**postgresql+psycopg2://username:password@host:port/database**'  
engine = sqlalchemy.**create\_engine**(url)

We can also include optional arguments inside a “create\_engine()” function. For example, we can add “-csearch\_path=schema\_name” to override the current session’s search path in PostgreSQL. This is equivalent to writing a query, “SET search\_path TO schema\_name”.

engine = sqlalchemy.create\_engine(params, **connect\_args**={'options': '**-csearch\_path=schema\_name**'})

**Run a SQL Query using SQLAlchemy**

Once we create a connection, we can interact with the SQL database in Python. Let’s start with the simplest query, “SELECT \* FROM table”.

from sqlalchemy.sql import **text**  
sql = '''  
 SELECT \* FROM table;  
'''  
**with engine.connect()**.execution\_options(**autocommit=True**) as conn:  
 query = conn.execute(**text**(sql))   
df = pd.DataFrame(query.**fetchall()**)

There are a few key functions we will use.

* **text():**SQLAlchemy allows users to use the **native SQL syntax** within Python with the function, “text()”. It would pass a textual statement to the SQL database mostly unchanged. Therefore, we can use the native SQL syntax, such as, **DELETE**, **UPDATE**, **INSERT**, **SELECT, Full-text Search** and others, within a Python framework.
* **engine.connect():**This function returns a SQL **Connection**object. By using it with a Python context manager (e.g., **with**statement), the “**Connection.close()**” function will be automatically involved at the end of the block of codes.
* **autocommit=True:**This optional argument inside the function, “.execution\_options()” allows us to turn on the auto-commit feature. That means we don’t need to write additional codes, such as, “connection.commit()” and “connection.rollback()”. “**connection.commit()**” would commit any changes to the SQL database while “**connection.rollback()**” would discard any changes. *One advantage of using autocommit is we have fewer lines of code and address potential issues of forgetting to commit changes.*
* **fetchall():** This function would return row objects, which can be integrated with Pandas to create a data frame.

In the following examples, we would **UPDATE, INSERT, DELETE**rows in a SQL table. The only different from **SELECT**is we write “conn.execute(text(sql))” instead of “query = conn.execute(text(sql))” because we’re not extracting a table.

# **Update** rows in a SQL table  
sql = '''  
 UPDATE table   
 SET col='abc'  
 WHERE condition;  
'''  
with engine.connect().execution\_options(autocommit=True) as conn:  
 **conn.execute(text(sql))**# **Insert** new rows in a SQL table  
sql = '''  
 INSERT INTO df  
 VALUES   
 (1, 'abc'),  
 (2, 'xyz'),  
 (1, 'abc');  
'''  
with engine.connect().execution\_options(autocommit=True) as conn:  
 **conn.execute(text(sql))**# **Delete** rows in a SQL table  
sql = '''  
 DELETE FROM df  
 WHERE condition;  
'''  
with engine.connect().execution\_options(autocommit=True) as conn:  
 **conn.execute(text(sql))**

Running SQL queries can very **flexible**in Python. We can set up a for-loop to run multiple SQL queries based on different conditions. For example:

For i in [value\_1, value\_2, value\_3, ...]:  
 if condition\_1:  
 sql = '''**sql\_query\_1**'''  
 elif condition\_2:  
 sql = '''**sql\_query\_2**'''  
 else:  
 sql = '''**sql\_query\_3**'''  
   
 with engine.connect().execution\_options(autocommit=True) as conn:  
 conn.execute(text(sql))

**Run Multiple SQL Queries**

Running multiple SQL queries in a single block is also straightforward. We just need to separate statements with **semicolons**. The simple implementation with SQLAlchemy makes it easy to interact with SQL in Python.

sql = '''  
 DROP TABLE IF EXISTS df;  
 CREATE TABLE df(  
 id SERIAL PRIMARY KEY,  
 salary integer  
 );  
 INSERT INTO df (salary)  
 VALUES   
 (400),  
 (200),  
 (3001);  
 SELECT \* FROM df;  
'''  
with engine.connect().execution\_options(autocommit=True) as conn:  
 query = conn.execute(text(sql))   
df = pd.DataFrame(query.fetchall())

**Store SQL Table in a Pandas Data Frame**

We’ve mentioned “fetchall()” function to save a SQL table in a pandas data frame. **Alternatively**, we can also achieve it using “**pandas.read\_sql**”. Since SQLAlchemy is integrated with Pandas, we can use its SQL connection directly with “con = conn”.

with engine.connect().execution\_options(autocommit=True) as conn:  
 df = pd.**read\_sql**(f"""SELECT \* FROM table\_name WHERE condition""", **con = conn**)

**Insert DataFrame into an Existing SQL Database**

To insert new rows into an existing SQL database, we can use codes with the native SQL syntax, INSERT, mentioned above. **Alternatively**, we can use “**pandas.DataFrame.to\_sql**” with an option of “ *if\_exists=‘append***’** ” to **bulk insert** rows to a SQL database. One benefit of this method is we can take full advantage of Pandas functionalities, such as, importing external data files and transforming raw data. So we can have a Pandas DataFrame that is compatible (e.g., having the same columns and data types as the SQL table) and ready to be inserted into an existing SQL database.

df = pd.read\_excel('sample.xlsx')  
with engine.connect().execution\_options(autocommit=True) as conn:  
 df.**to\_sql**('table\_name', con=conn, **if\_exists=**'**append**', index= False)

**Create a New SQL Database**

“**pandas.DataFrame.to\_sql**” also works on creating a new SQL database. As you can see from the following example, we import an external data from a excel spreadsheet and create a new SQL table from the pandas DataFrame.

from **sqlalchemy.types** import Integer, Text, String, DateTimedf = pd.read\_excel('sample.xlsx')  
df.**to\_sql**(  
 "table\_name",   
 con = engine,  
 **if\_exists = "replace"**,  
 schema='shcema\_name',   
 index=False,  
 chunksize=1000,  
 dtype={  
 "col\_1\_name": Integer,  
 "col\_2\_name": Text,  
 "col\_3\_name": String(50),  
 "col\_4\_name": DateTime  
 }  
)

There are a few important arguments we need to specify with “**to\_sql()**” function in order to create a new SQL table properly.

* **if\_exists**: This argument would indicate what to do if a table with the name “table\_name” already exists in the database. Passing “**replace**” would drop all rows in the existing table and replace it with the current pandas data frame. Passing “**append**”, as mentioned above, would only append the pandas data frame into the existing SQL table.
* **schema**: This argument would take the schema name where you would save your new SQL table. It is not required if you’ve already specified the schema name in the connection.
* **index**: This argument would indicate whether we would create a column in the new SQL table for DataFrame’s index.
* **chuncksize**: This argument would specify the number of rows in each batch to be inserted at a time. By default, all rows will be written at once.
* **dtype**: This argument would specify the datatype of columns in the new SQL table. The datatypes we used are from **sqlalchemy.types.**