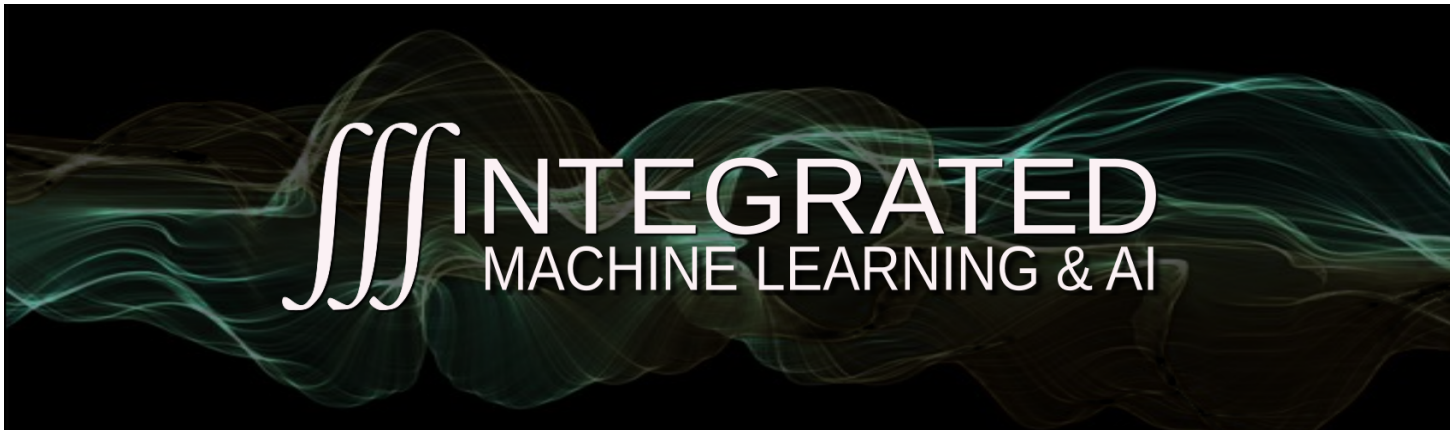# SQL Thru Python

A Python class that connects with SQL Server and uses SQLAlchemy (on top of pyodbc) and Pandas to streamline SQL operations thru Python ... Thom Ives and Ghaith Sankari

All of the code explained in this PDF is available from Thom's SQL_Thru_Python repository on DagsHub.

Overview

Please consider this as a starter class for your own SQL thru Python operations. Take it and make it better for your specific needs. Most of the methods in this class are literally convenience methods. Why do we say this? Because truly, all other methods and more could be accomplished using the last method `general_sql_comm`. In fact, when we bury our SQL routines in Python classes for reuse, we create many more convenience routines and procedures to format our data coming from, and going into, SQL tables in various databases. So, again, consider this a set of starter methods for a class you will customize for your own applications.

The skilled SQL programmer might ask, "Why not just make a set of **Stored SQL Procedures**? We couldn't agree more! We are simply doing this on the Python side. Why? To make it part of our Python automations, which include other Python automations for our applications.

Imports

First, we import pandas, sqlalchemy, and, we find this next part essential, `import URL` from `sqlalchemy.engine`. We'll explain why soon.

Next, we view the class definition with only the method names shown.

```python
import pandas as pd
import sqlalchemy
from sqlalchemy.engine import URL

class DB_Table_Ops:
>    def __init__(self,

>    def table_exists(self, table_name):

>    def create_table(self, schema_str):

>    def drop_table(self, table_name):

>    def insert_df_to_table(self, df, table_name):

>    def query_to_df(self, sql_comm):

>    def general_sql_comm(self, sql_comm):
```

The init Method

The `__init__` method has default values for: 1) driver, 2) server, 3) database, 4) user, and 5) password.

You can of course pass in values other than the default. **Only use this type of code in your safely protected backend shared with other trusted developers.**

Using the parameters of the `__init__` method, we then build up a connection_string. The connection_string is input to the `URL.create` method along with other parameters that are needed as described in SQL Alchemy documentation. The `URL.create` method was found to be the least cumbersome way to create a reliable url connection string that would work.

The `connection_url` is generated and passed as an input to `sqlalchemy.create_engine`, which returns connection engine information that is assigned to the class's `self.engine` attribute.

```python
def __init__(self,
             driver='{ODBC Driver 17 for SQL Server}',
             server='your server name',
             database='your database name',
             user='your user name',
             pw='your password'):

    connection_string = f'DRIVER={driver};SERVER={server};'
    connection_string += f'DATABASE={database};'
    connection_string += f'UID={user};PWD={pw}'

    # create sqlalchemy engine connection URL
    connection_url = URL.create(
        "mssql+pyodbc",
        query={"odbc_connect": connection_string})

    self.engine = sqlalchemy.create_engine(connection_url)
```

The table_exists Method

The table_exists method simply checks to see if the provided table name exists in our database. The fixed query string (qs) shown will return a list of all table names in our database. If it finds an instance of the table name, it returns `True`. If not, it returns `False`.

```python
def table_exists(self, table_name):
    qs = '''select schema_name(t.schema_id) as the_schema_names,
                t.name as the_table_names
                from sys.tables t
                order by the_schema_names,
                the_table_names;'''

    with self.engine.connect() as conn:
        cursor = conn.execute(sqlalchemy.text(qs))
        table_exists = [t for t in cursor if table_name == t[1]]

        return bool(table_exists)
```
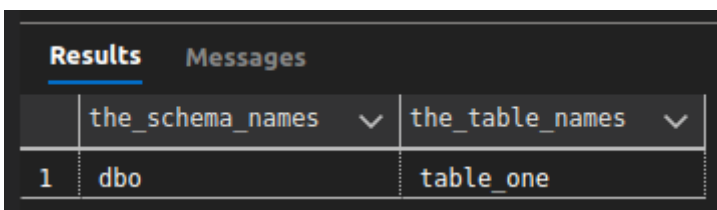
If you were to run the query string, qs, in **SQL Server Management Studio** or **Azure Data Studio**, you'd see a list of existing tables. For operations we will soon run, you'd see the following.

	the_schema_names	the_table_names
1	dbo	table_one

In case you are not familiar with context managers, consider the following

```
with self.engine.connect() as conn:
```

The Python `with` statement is followed by a Python command and is assigned an alias using `as`. The alias of `conn` is then used below the context manager in the indented lines of code.

Context managers are great for helping us write clean and concise code. They are implemented using the `with` statement. When the block of code indented under a with statement is done, the connection is closed. We encourage you to study context managers when you can. You can even write your own. You will see the above context manager in most of the other methods of this class.

The list comprehension

```
table_exists = [t for t in cursor if table_name == t[1]]
```

will be empty if `table_name` is not found, and will have only one element if `table_name` is found. Finally, the list is type cast as a boolean. If the list is empty, it will return `False`. If `table_name` was found among the table names, it will return `True`.

The create_table Method

The create_table method simply takes a schema string as the only input. We extract the table name from this string. Then we call the `table_exists` method. If the table already exists, we simply return to the calling code. If the table does not yet exist, Then, using our context manager, we connect to the engine, and we create the table. Once this code is complete, the context manager closes the engine connection.

```
def create_table(self, schema_str):
    table_name = schema_str.split()[2]
    if not self.table_exists(table_name):
        with self.engine.connect() as conn:
            conn.execute(sqlalchemy.text(schema_str))
```

The drop_table Method

The `drop_table` method drops a table. If there is no table with `table_name`, we return to the calling code. If it does exist, we drop that table within our same context manager.

```
def drop_table(self, table_name):
    if self.table_exists(table_name):
        sql_comm = f'''DROP TABLE {table_name}'''
        with self.engine.connect() as conn:
            conn.execute(sqlalchemy.text(sql_comm))
```

The insert_df Method

The `insert_df_to_table` method shows us the amazing elegance of SQLAlchemy and Pandas working together.

```
def insert_df_to_table(self, df, table_name):
    if type(df) is dict:
        df = pd.DataFrame(data=df)
    df.to_sql(table_name, self.engine,
                if_exists="append", index=False)
```

First, note the code

```
if type(df) is dict:
    df = pd.DataFrame(data=df)
```

There may be times that you'd prefer to pass in a python dictionary that is in the same format you'd receive from `df_dictionary = df.to_dict()`. Just in case such a dictionary is passed in, we want to first convert it to a pandas dataframe to exploit the power of SQLAlchemy and Pandas working together.

The Pandas `to_sql` method, for dataframe class instances, uses our engine connection and uses its own context manager. We also pass in `table_name`, `if_exists="append"`, and `index=False`. Try other parameter values to see the differences. We've simply found these parameters to provide the greatest consistency for adding data to tables and subsequently querying data from tables.

If the table exists, the `if_exists="append` ensures that new data in the dataframe is appended to the end of the table. We personally don't want this Pandas method to use the dataframe's indices for the SQL table's indices. We prefer to let SQL control the indices in accordance with the table schema during `create table`.

The query_to_df Method

When we want to query data from a SQL table, we use our `query_to_df` method.

```python
    def query_to_df(self, sql_comm, index_col="ident", return_dictionary=False):
        with self.engine.connect() as conn:
            df = pd.read_sql_query(sql_comm, conn, index_col=index_col)

            if return_dictionary:
                return df.to_dict()
            else:
                return df
```

We simply pass in our `sql_comm`, which would be some specific SQL query, and we also have two parameters with default values: `index_col="ident"`, and `return_dictionary=False`. We prefer to always name our indentity column in our SQL tables `indent`, but there may be times that this is not desirable. Also, there maybe times that we want to return a dictionary of a dataframe instead of a dataframe.

Our connection `conn` is opened by the context manager. The Pandas' `read_sql_query` method is called using the `sql_comm`, the engine connection `conn`, and our index_col name as a string. This method returns a dataframe, or a dictionary (depending on the `return_dictionary` parameter). Then we return the desired data object to the calling code. The engine connection is closed when code within the context manager is complete.

The general_sql_comm Method

The last method of our class is `general_sql_comm`, which receives a SQL command.

```python
    def general_sql_command(self, sql_comm):
        with self.engine.connect() as conn:
            cursor = conn.execute(sqlalchemy.text(sql_comm))
            try:
                return [list(t) for t in cursor]
            except:
                pass
```

We use a context manager again to control opening and closing the connection. Once connected we pass in our general SQL command, `sql_comm` for execution by our database. Consider how many lines of code were saved by our context manager invoked using `with self.engine.connect() as conn:`. We hope you will fully exploit context management in your Python code moving forward.

Testing Our Class Methods

Let's now start a separate Python file to import pandas and our `DB_Table_Ops` from the Python file containing it. In our case we named it `Py_Sql_Alchemy_Class.py` and placed it in the same directory.

```python
import pandas as pd
from Py_Sql_Alchemy_Class import DB_Table_Ops


d = {'value_1': [1, 2], 'value_2': [3, 4]}
df = pd.DataFrame(data=d)

schema_str = '''CREATE TABLE table_one (
    ident int IDENTITY(1,1) PRIMARY KEY,
    value_1 int NOT NULL,
    value_2 int NOT NULL);'''

update_cmd = '''UPDATE table_one
    SET value_1 = 7, value_2 = 10
    WHERE ident = 3;'''

dbto = DB_Table_Ops()

dbto.drop_table('table_one')
print(f'table_one exists: {dbto.table_exists("table_one")}')
dbto.create_table(schema_str)
print(f'table_one exists: {dbto.table_exists("table_one")}')

if True:
    dbto.insert_df_to_table(df, 'table_one')
    dbto.insert_df_to_table(df, 'table_one')
    dbto.general_sql_command(update_cmd)

query_string = 'SELECT * FROM table_one'
print('\nData from table one:')
print(dbto.query_to_df(query_string))
print()

dbto.drop_table('table_one')
print(f'table_one exists: {dbto.table_exists("table_one")}')
```

We do the following operations:

1. Create a simple dictionary
2. Convert the simple dictionary to a dataframe
3. Define a SQL command as a string to create a specific type of table schema
4. Define a SQL command that can perform an update to a table
5. Create an instance of `DB_Table_Ops` and name it `dbto`

6. Drop `table_one` if it exists and report the state of it
7. Create `table_one` using the predefined schema string and report the state of it
8. Insert our predefined dataframe into `table_one` twice, and then update it
9. Query the contents of `table_one` and return the data in a Pandas dataframe
10. Drop `table_one` and check its existence

The output from this script is ...

```
table_one exists: False
table_one exists: True

Data from table one:
       value_1  value_2
ident
1            1        3
2            2        4
3            7       10
4            2        4

table_one exists: False
```

Closing

We encourage you to play with this code. Make changes. Make it more appropriate for your specific needs. We've refactored this many times ourselves now, but seek to make it still better at least for yourself. And most of all, enjoy!

Thom Ives and Ghaith Sankari