PER SCHOLAS

GLAB - 386.4.9 - Read Data From the Database Into a Dataframe

---

Lab Overview:

This lab demonstrates how to connect with a MySQL database using the **SQLAlchemy** module, read data from the database into a DataFrame, perform basic EDA, group by a column, and aggregate data.

Lab Objective:

By the end of this lab, learners should be able to:

- Install and configure the SQLAlchemy module.
- Configure the MySQL database with a Pandas project.
- Use the Pandas read_sql() function to read data from the database into a DataFrame.
- Perform exploratory data analysis (EDA),
- Demonstrate grouping and aggregation using the MySQL database and table.

Prerequisites:

- For this lab, you must have a **"classicmodels"** database. If you do not have a **'classicmodels'** database setup, click here to download the database script file.
- Python environment.
- Python-related IDE (e.g., Anaconda, Jupyter Notebook).
- NumPy library installed (can be installed via pip or Conda).

Submission:

- Submit your completed lab using the Start Assignment button on the Assignment page in Canvas.
  - if you are using notebook then, all tasks should be written and submitted in a single notebook file, for example: **(your_name_labname.ipynb).**
  - if you are using python script file, all tasks should be written and submitted in a single python script file for example: **(your_name_labname.py).**
  - Add appropriate comments and any additional instructions if required.

Introduction:

Pandas uses **SQLAlchemy**, a popular Python library, to interact with SQL databases. **SQLAlchemy.** In turn, you will need a **database driver** to connect to MySQL.

Reference Link: **SQLAlchemy**

You can install them using the following pip commands:

> Note: **Do not use the Google Colab notebook for this lab because of an issue with mysqlclient.**
>
> The mysqlclient package **may not install easily on Google Colab** because it requires system-level dependencies (libmysqlclient-dev).
>
> Use a local IDE (e.g., Notepad, Sublime Text) or a notebook environment (e.g., Jupyter Notebook, VS Code Notebook).

```
## Run below from the command line
 # install sqlalchemy
pip install SQLAlchemy

 # install mysql driver
 pip install mysql-connector-python

 # use pymysql model:
 pip install pymysql

 #Install Additional model for SQL client
 pip install mysqlclient
```

Instructions:

- **SQLAlchemy** requires us to **create a database engine**. Then we can use this engine to get database connections whenever we want to run SQL queries.
- The below code uses SQLAlchemy's **create_engine()** method to get a new database **engine** for `**classicmodels**` database. Then we will use the **read_sql()** function to execute a query and store the data in a Pandas DataFrame.
- In the below examples, we will use two tables named **products** and **orderdetails.**

Let's get data from the `products` table and the `orderdetails` table.

```
import pandas as pd
from sqlalchemy import create_engine,text
import mysql.connector as dbconnect
# The create_engine() function will create your connection
engine =
create_engine("mysql+mysqldb://root:password@localhost:3305/classi
cmodels")
sql_query_order = """ SELECT orderNumber, productCode,priceEach,
orderLineNumber, quantityOrdered FROM orderdetails; """

 SQL_Query_product = """ SELECT * FROM products """;
 with engine.connect() as my_conn:
    # Use pandas read_sql() to read data from the database into a
dataframe.
      #Using Order table
     my_data = pd.read_sql(text(SQL_Query_product),my_conn)
     #print all records from table
     print(my_data)
```

1. **Print 10 records from the dataframe using pandas.head() function.**

```
    print(my_data.head(10))
```

2. **Print only specific columns using the Pandas square [ ] attribute.**

```
print(my_data[['productCode','productName']].head(10))
```

3. We can specify the index column using index_col parameter as shown below.

```
products_df = pd.read_sql(text(SQL_Query_product),my_conn,
index_col ='productCode')

print(products_df)
```

4. Perform Exploratory Data Analysis (EDA).

```
 print("\nBasic Statistics:")

 print(products_df.describe())
```

5. Check data types.

```
print(products_df.dtypes)
```

6. Find the number of rows and columns.

The number of rows and columns in a DataFrame can be identified using the ***'shape'*** attribute of the Panda Dataframe. It returns a tuple (row, column) and can be indexed to get only rows or only columns as output.

```
print(products_df.shape) # Get the number of rows and columns
print(products_df.shape[0]) # Get the number of rows only
print(products_df.shape[1]) # Get the number of columns only
```

7. Check for missing values.

```
print("\nMissing Values:")
print(products_df.isnull().sum())
```

Note: You can perform additional EDA or analysis based on your specific requirements.

8. Grouping and Aggregations

Example: Group by 'productLine' and calculate the total “quantityInStock” and average price for each productLine:

```
grouped_df = products_df.groupby('productLine').agg({'quantityInStock': 'sum', 'buyPrice': 'mean'}).reset_index()
    print("\nGrouped Data:")
    print(grouped_df)
```

*Note: The **reset_index()** function is used to move the grouped columns back to regular columns*

Using ‘orderdetails’ Table

9. Find the total amount for each order.

In the following example, a new column, **'totalCost,'** is created by multiplying the **'buyPrice'** and **'quantityOrdered'** columns element-wise. This results in a DataFrame with the additional **'totalCost'** column, which is then grouped by the **'orderNumber'** column.

```
 orders_prod_df =
pd.read_sql(text(sql_query_order),my_conn)
     print("Sample of the 'orders' DataFrame:")
     print(orders_prod_df.head())

     orders_prod_df['totalCost'] =
orders_prod_df['priceEach'] *
orders_prod_df['quantityOrdered']
     # Group by 'orderNumber' and sum 'totalCost' for each
group
     grouped_df =
orders_prod_df.groupby('orderNumber')['totalCost'].sum().re
set_index()
     print(grouped_df)
```

Note: *You can replace* ***'totalCost'*** *with any other column name that makes sense for your specific use case. The key is to use the * operator to perform element-wise multiplication between the two columns.*

Here is the complete code example:

```
import pandas as pd
 from sqlalchemy import create_engine,text
 import mysql.connector as dbconnect
 # The create_engine() function will create your connection
 engine =
create_engine("mysql+mysqldb://root:password@localhost:3306/classi
cmodels")
 #my_conn=conn.connect()
 sql_query_order =""" SELECT orderNumber, productCode, priceEach,
orderLineNumber, quantityOrdered FROM orderdetails; """
 SQL_Query_product = """SELECT * FROM products""";

 with engine.connect() as my_conn:

    # Use pandas read_sql() to read data from the database into a
Dataframe
     my_data = pd.read_sql(text(SQL_Query_product),my_conn)
     #print all records from table
     print(my_data)

     print(my_data.head(10))

     products_df =
pd.read_sql(text(SQL_Query_product),my_conn,index_col
='productCode')
     print(products_df)

     print(products_df[['buyPrice','productName']].head(10))

     print("\nBasic Statistics:")
     print(products_df.describe())
```

```
      print(products_df.dtypes)

      print(products_df.shape) # Get the number of rows and
columns.
      print(products_df.shape[0]) # Get the number of rows only.
      print(products_df.shape[1]) # Get the number of columns only

      print("\nMissing Values:")
      print(products_df.isnull().sum())

      # Grouping and Aggregations
     # Example: Group by 'productLine' and calculate the total
quantityInStock and average price for each productLine

                grouped_df =
products_df.groupby('productLine').agg({'quantityInStock': 'sum',
'buyPrice': 'mean'}).reset_index()
                print("\nGrouped Data:")
                print(grouped_df)



     #Using Order table
      orders_prod_df = pd.read_sql(text(sql_query_order),my_conn)
      print("Sample of the 'orders' DataFrame:")
      print(orders_prod_df.head())

      orders_prod_df['totalCost'] = orders_prod_df['priceEach'] *
orders_prod_df['quantityOrdered']
      # Group by 'orderNumber' and sum 'totalCost' for each group
      grouped_df =
```

```
orders_prod_df.groupby('orderNumber')['totalCost'].sum().reset_ind
ex()
     print(grouped_df)
```