16_SQL para analistas de datos

June 18, 2025

Creado por:

Isabel Maniega

0.1 Ejecutar comandos SQL fundamentales para crear, leer, actualizar y eliminar datos en tablas de bases de datos. Establecer conexiones a bases de datos usando Python.

0.2 SQLAlchemy

¿Qué es una base de datos?

Hoy en día, las aplicaciones sociales como Facebook, Twitter e Instagram son muy populares. Cada día, muchas personas crean nuevas cuentas y los usuarios existentes agregan o comparten diferentes contenidos. Seguramente habrás notado que los datos enviados a esas aplicaciones siguen estando disponibles después de unos días o incluso años. ¿Sabes cómo es posible?

Los datos son simplemente información sobre los usuarios, el contenido de sus publicaciones y comentarios. Los datos son todo tipo de información que podemos enviar a la aplicación. Durante el registro, envías a la aplicación los datos de tu cuenta, que consisten en la dirección de correo electrónico, el nombre de usuario y la contraseña, y cuando agregas nuevas publicaciones envías contenido que será visible para otros usuarios. Los datos enviados deben guardarse en algún lugar al que se pueda acceder fácilmente. Este lugar es una base de datos, que es un conjunto de información almacenada en un disco en un sistema informático. El acceso a la base de datos es posible gracias a un sistema de gestión de bases de datos.

El sistema de gestión de bases de datos (DBMS) es el software responsable de:

- crear una estructura de base de datos;
- insertar, actualizar, eliminar y buscar datos;
- garantizar la seguridad de los datos;
- gestión de transacciones;
- garantizar el acceso simultáneo a los datos para muchos usuarios;
- permitir el intercambio de datos con otros sistemas de bases de datos.

Existen muchos sistemas de gestión de bases de datos gratuitos y pagos en el mercado. Los más populares son:

Gratuitos	Pagos
MySQL, PostgreSQL, SQLite	Oracle Database, Microsoft SQL Server, IBM DB2

Más adelante en este curso, nos centraremos principalmente en el sistema SQLite. Lo utilizarás para crear tu primera aplicación de base de datos Python. ¿Estás listo? ¡Vamos!

¿Qué es SQLAlchemy y por qué deberías usarlo?

SQLAlchemy es el conjunto de herramientas SQL de Python que permite a los desarrolladores acceder y gestionar bases de datos SQL con el lenguaje de dominio Python. Puedes escribir una consulta en forma de cadena o encadenar objetos Python para consultas similares. Trabajar con objetos proporciona flexibilidad a los desarrolladores y les permite crear aplicaciones de alto rendimiento basadas en SQL.

En palabras sencillas, permite a los usuarios conectar bases de datos utilizando lenguaje Python, ejecutar consultas SQL mediante programación basada en objetos y agilizar el flujo de trabajo.

Cómo trabajar con una base de datos SQLAlchemy usando Python

Es bastante fácil instalar el paquete y empezar a codificar.

Puedes instalar SQLAlchemy con el gestor de paquetes de Python (pip):

```
pip install sqlalchemy
```

sqlalchemy – creación de una base de datos

La base de datos SQLAlchemy se guarda en un único archivo. Cada archivo, independientemente del sistema operativo utilizado, tiene su ubicación (una ruta a un espacio de disco específico). Cuando creas una base de datos, puedes crearla en tu directorio de trabajo actual o en cualquier otra ubicación. Para crear una base de datos, utiliza el método `create_engine` que proporciona `sqlite` creando la base de datos en el archivo database.db:

```
from sqlalchemy as db

engine = db.create_engine("sqlite:///database.db")

conn = engine.connect()
```

El método `connect` devuelve la representación de la base de datos como un objeto `Connection`. En el ejemplo anterior, el método solo toma el nombre de la base de datos como argumento. Esto significa que la base de datos se creará en el mismo directorio que el script que desea acceder a ella.

Recuerda que el método `connect` crea una base de datos solo si no puede encontrar una base de datos en la ubicación indicada. Si existe una base de datos, SQLite se conecta a ella.

Algunas palabras sobre SQL

Ya has aprendido a crear una base de datos en Python usando el módulo `sqlalchemy`. Ahora es el momento de analizar cómo podemos crear su estructura. Para ello, necesitaremos algunos conocimientos de SQL.

SQL es un **lenguaje de consulta estructurado** para crear, modificar y gestionar bases de datos relacionales. Lo utilizan los sistemas de gestión de bases de datos más populares, como MySQL, PostgreSQL y nuestro favorito SQLite. El lenguaje SQL fue desarrollado en los años 70 por IBM. A lo largo de los años, SQL ha sido modificado por muchas empresas que lo han implementado en sus productos. Por lo tanto, se hizo necesario introducir un estándar que estandarizara su sintaxis.

sqlalchemy – creación de tablas

En primer lugar, crearemos una nueva base de datos llamada “database.db”. create_engine creará una nueva base de datos automáticamente si no existe ninguna base de datos con el mismo nombre. Por tanto, crear y conectar son bastante similares.

A continuación, conectaremos la base de datos y crearemos un objeto de metadatos.

Utilizaremos la función Tabla de SQLAlchmy para crear una tabla llamada “Task”.

Consta de columnas:

- Id: Entero y clave principal
- name: Cadena y no anulable
- priority: Entero y no anulable

Hemos creado la estructura de la tabla. Vamos a añadirlo a la base de datos utilizando `metadata.create_all(engine)`.

```
engine = db.create_engine('sqlite:///database.db')
conn = engine.connect()
metadata = db.MetaData()

Student = db.Table('Task', metadata,
              db.Column('Id', db.Integer(),primary_key=True),
              db.Column('name', db.String(255), nullable=False),
              db.Column('priority', db.Integer(), nullable=False),
              )

metadata.create_all(engine)
```

sqlalchemy – inserción de datos

Para insertar una línea utilizaremos `Text` para poder pasar el comando SQL para insertar un dato, en la tabla, con ayuda de `execute` ejecutaremos el comando en la base de datos, la estructura `with` gestiona automáticamente el cierre de la base de datos:

```
from sqlalchemy import text

with engine.connect() as conn:

    rs = conn.execute(text("INSERT INTO Task (name, priority) VALUES ('My first task', 1)"))

    conn.commit()
```

[1]:
```
# pip install sqlalchemy
```

[1]:
```
import sqlalchemy as db

engine = db.create_engine('sqlite:///./files/database.db')
conn = engine.connect()
metadata = db.MetaData()

Task = db.Table('Task', metadata,
```

```
              db.Column('Id', db.Integer(),primary_key=True),
              db.Column('name', db.String(255), nullable=False),
              db.Column('priority', db.Integer(), nullable=False),
              )

metadata.create_all(engine)
```

---

sqlalchemy – el método `execute()`

Realizar muchas consultas no es muy eficiente cuando podemos utilizar solo una que realice la misma tarea. Imagine una situación en la que desea agregar tres tareas a la base de datos. Si utilizamos el método de ejecución, tendríamos que realizar tres consultas independientes.

Esta no es una buena práctica. Afortunadamente, el objeto Cursor nos ofrece un método llamado `execute`. Observe el código en el editor.

El método `execute` le permite insertar varios registros a la vez. Como argumento, acepta una sentencia SQL y una matriz que contiene cualquier número de diccionarios.

[2]:
```
query = db.insert(Task)
values_list = [{"name": 'My first task', "priority": 1},
               {"name": 'My second task', "priority": 5},
               {"name": 'My third task', "priority": 10},]
Result = conn.execute(query,values_list)
conn.commit()
conn.close()
```

sqlalchemy – lectura de datos

Hasta ahora, no hemos mostrado ninguna información en la pantalla sobre las tareas insertadas. Es hora de cambiar eso. Veamos qué hay en nuestra base de datos. Primero necesitaremos la sentencia SQL adecuada, llamada `SELECT`.

La sentencia `SELECT` permite leer datos de una o más tablas. Su sintaxis es la siguiente:

```
SELECT column FROM table_name;
```

o

```
SELECT column1, column2, column3, …, columnN FROM table_name;
```

o

```
SELECT * FROM table_name;
```

En la primera variante, decidimos leer los valores guardados en una sola columna. Si quisiéramos leer únicamente los nombres de las tareas guardadas en la tabla de tareas, podríamos utilizar la siguiente consulta:

```
SELECT name FROM tasks;
```

La segunda variante permite leer valores de más columnas. Si queremos leer los nombres de las tareas y sus prioridades, podemos utilizar la siguiente consulta:

`SELECT name, priority FROM tasks;`

Si no tenemos ningún requisito específico, podemos leer los valores de todas las columnas:

`SELECT * FROM tasks;`

La última variante mostrará los valores guardados en las columnas id, name y priority.

sqlalchemy – lectura de datos

Probablemente no te sorprenda saber que la lectura de datos guardados en la base de datos se realiza con el conocido objeto `execute`. Después de llamar al método de ejecución con la instrucción `SELECT` adecuada, el objeto `rs` se trata como un iterador. Observa el código en el editor.

Resultado:

```
(1, 'My first task', 1)
(2, 'My second task', 5)
(3, 'My third task', 10)
```

La variable fila en cada iteración toma una fila en forma de tupla. El acceso a las columnas individuales se realiza mediante un índice, por ejemplo, print (row [0]) mostrará los valores guardados en la columna id.

[3]:
```
from sqlalchemy import text

with engine.connect() as conn:

    rs = conn.execute(text('SELECT * FROM Task'))

    for row in rs:
        print(row)
```

```
(1, 'My first task', 1)
(2, 'My second task', 5)
(3, 'My third task', 10)
```

sqlalchemy – lectura de datos

Si no desea tratar el objeto `rs` como un iterador, puede utilizar su método llamado `fetchall`. El método `fetchall` recupera todos los registros (aquellos que aún no se han recuperado del resultado de la consulta). Observe el código en el editor.

Resultado:

```
(1, 'My first task', 1)
(2, 'My second task', 5)
(3, 'My third task', 10)
```

El método `fetchall` es menos eficiente que el iterador, porque lee todos los registros en la memoria y luego devuelve una lista de tuplas. Para pequeñas cantidades de datos, no importa, pero si su tabla contiene una gran cantidad de registros, esto puede causar problemas de memoria.

NOTA: El método `fetchall` devuelve una lista vacía cuando no hay filas disponibles.

[4]:
```
with engine.connect() as conn:

    rs = conn.execute(text('SELECT * FROM Task'))
    rows = rs.fetchall()
    for row in rows:
        print(row)
```

```
(1, 'My first task', 1)
(2, 'My second task', 5)
(3, 'My third task', 10)
```

Otra forma de mostrar los datos es usar el método `select()` que viene incorporado con el sqlalchemy:

[5]:
```
with engine.connect() as conn:
    query = Task.select()
    output = conn.execute(query)
    print(output.fetchall())
```

```
[(1, 'My first task', 1), (2, 'My second task', 5), (3, 'My third task', 10)]
```

sqlalchemy – lectura de datos

Además del iterador y el método `fetchall`, el objeto `rs` proporciona un método muy útil llamado `fetchone` para recuperar el siguiente registro disponible. Observa el código en el editor.

Resultado:

```
(1, 'My first task', 1)
(2, 'My second task', 5)
```

NOTA: El método `fetchone` devuelve None si no hay datos para leer.

[6]:
```
with engine.connect() as conn:

    rs = conn.execute(text('SELECT * FROM Task'))
    row = rs.fetchone()
    print(row)
    row = rs.fetchone()
    print(row)
```

```
(1, 'My first task', 1)
(2, 'My second task', 5)
```

sqlalchemy – actualización de datos

Cada una de las tareas creadas tiene su propia prioridad, pero ¿qué pasa si decidimos que una de ellas debe realizarse antes que las demás? ¿Cómo podemos aumentar su prioridad? Tenemos que utilizar la sentencia SQL llamada `UPDATE`.

La sentencia `UPDATE` se utiliza para modificar registros existentes en la base de datos. Su sintaxis es la siguiente:

```
UPDATE table_name
```

```
SET column1 = value1, column2 = value2, column3 = value3, …, columnN = valueN
WHERE condition;
```

Si queremos establecer la prioridad en 20 para una tarea con ideigual a 1, podemos utilizar la siguiente consulta:

```
UPDATE Task SET priority = 20 WHERE id = 1;
```

NOTA: Si olvida la cláusula `WHERE`, se actualizarán todos los datos de la tabla.

Como antes, ejecutamos todas las sentencias SQL utilizando el método de ejecución. Observe el código en el editor.

[6]:
```
with engine.connect() as conn:

    rs = conn.execute(text('UPDATE Task SET priority = 20 WHERE id = 1'))
    row = conn.execute(Task.select()).fetchone()
    print(row)
    conn.commit()
```

```
(1, 'My first task', 20)
```

Otra forma es usar el método `update()`:

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

    query = Task.update().where(Task.columns.Id == 1).values(priority=20)
    output = conn.execute(query)
```

sqlalchemy – eliminación de datos

Luego de completar una tarea, queremos eliminarla de nuestra base de datos. Para ello, debemos utilizar la sentencia SQL denominada `DELETE`:

```
DELETE FROM table_name WHERE condition;
```

Veamos cómo se vería eliminar la tarea con `id = 1`:

```
DELETE FROM Task WHERE id = 1;
```

NOTA: Si olvida la cláusula `WHERE`, se eliminarán todos los datos de la tabla.

[4]:
```
with engine.connect() as conn:

    rs = conn.execute(text('DELETE FROM Task WHERE id = 1'))
    row = conn.execute(Task.select()).fetchone()
    print(row)
    conn.commit()
```

```
(2, 'My second task', 5)
```

Otra forma es usar el método `delete()`:

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

    query = Task.delete().where(Task.columns.Id == 1)
```

```
output = conn.execute(query)
```

Excepciones sqlalchemy

La jerarquía de excepciones está definida por DB-API 2.0 (PEP 249).

exception sqlalchemy.InterfaceError

Se genera una excepción para errores que están relacionados con la interfaz de la base de datos en lugar de con la base de datos en sí.

exception sqlalchemy.DatabaseError

Excepción generada para errores relacionados con la base de datos. Sirve como excepción base para varios tipos de errores de base de datos. Solo se genera de forma implícita a través de las subclases especializadas. DatabaseError es una subclase de Error.

exception sqlalchemy.DataError

Excepción generada para errores causados por problemas con los datos procesados, como valores numéricos fuera de rango y cadenas demasiado largas. DataError es una subclase de DatabaseError.

exception sqlalchemy.OperationalError

Excepción generada para errores relacionados con el funcionamiento de la base de datos y no necesariamente bajo el control del programador. Por ejemplo, no se encuentra la ruta de la base de datos o no se pudo procesar una transacción. OperationalError es una subclase de DatabaseError.

exception sqlalchemy.IntegrityError

Excepción generada cuando se afecta la integridad relacional de la base de datos, p. ej. Se produce un error en la comprobación de una clave externa. Es una subclase de DatabaseError.

exception sqlalchemy.InternalError

Se genera una excepción cuando SQLite encuentra un error interno. Si se genera, puede indicar que hay un problema con la biblioteca SQLite en tiempo de ejecución. InternalError es una subclase de DatabaseError.

exception sqlalchemy.ProgrammingError

Excepción generada por errores de programación de la API de sqlite3, por ejemplo, proporcionar la cantidad incorrecta de enlaces a una consulta o intentar operar en una conexión cerrada. ProgrammingError es una subclase de DatabaseError.

exception sqlalchemy.NotSupportedError

Excepción generada en caso de que la biblioteca SQLite subyacente no admita un método o una API de base de datos. Por ejemplo, establecer deterministic en True en create_function(), si la biblioteca SQLite subyacente no admite funciones deterministas. NotSupportedError es una subclase de DatabaseError.

[7]:
```
import sqlalchemy as db
from sqlalchemy import exc

engine = db.create_engine('sqlite:///./files/database.db')
```

```
conn = engine.connect()
try:
    metadata = db.MetaData()

    Task = db.Table('Task', metadata,
                    db.Column('Id', db.Integer(),primary_key=True),
                    db.Column('name', db.String(255), nullable=False),
                    db.Column('priority', db.Integer(), nullable=False),
                    )

    metadata.create_all(engine)
except exc.OperationalError as e:
    # si ocurre un error al crear la tabla nos mostrará el siguiente error:
    print(str(e))
conn.close()
```

0.3 pyodbc

pyodbc es una biblioteca de Python que permite a los programas Python interactuar con bases de datos mediante ODBC (Open Database Connectivity), una API estándar para acceder a sistemas de gestión de bases de datos (SGBD). Proporciona una forma potente y eficiente de ejecutar consultas SQL, recuperar resultados y realizar otras operaciones con bases de datos.

[2]:
```
# pip install pyodbc
```

Crear la base de datos

En linux es necesario instalar:

```
sudo apt install libsqliteodbc
sudo apt install libodbc2
```

Para este caso usaremos tambien como conector a una base de datos tipo SQLite, donde se lo indicaremos en el método `connect()`:

[19]:
```
import pyodbc

conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
try:
    c.execute('''CREATE TABLE IF NOT EXISTS tasks (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    priority INTEGER NOT NULL
    );''')
# también incluye las excepciones:
except pyodbc.Error as ex:
  print("An error occurred:", ex)
conn.commit()
```

```
conn.close()
```

Insertar los datos

Para insertar varios datos en la tabla usamos `executemany()`:

[9]:
```
conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
tasks = [
    ('My first task', 1),
    ('My second task', 5),
    ('My third task', 10),
]
c.executemany('INSERT INTO tasks (name, priority) VALUES (?,?)', tasks)
conn.commit()
conn.close()
```

Leer los datos

[11]:
```
conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
for row in c.execute('SELECT * FROM tasks'):
    print(row)
conn.close()
```

```
(1, 'My first task', 1)
(2, 'My second task', 5)
(3, 'My third task', 10)
```

También posee el método `fetchall` para leer todos los datos que tenemos como un único objeto:

[12]:
```
conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
c.execute('SELECT * FROM tasks')
rows = c.fetchall()
for row in rows:
    print(row)
conn.close()
```

```
(1, 'My first task', 1)
(2, 'My second task', 5)
(3, 'My third task', 10)
```

De igual modo tenemos el método `fetchone` para leer dato a dato:

[13]:
```
conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
c.execute('SELECT * FROM tasks')
row = c.fetchone()
print(row)
row = c.fetchone()
```

```
print(row)
conn.close()
```

```
(1, 'My first task', 1)
(2, 'My second task', 5)
```

Actualizar los datos

[16]:
```
conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
c.execute('UPDATE tasks SET priority = ? WHERE id = ?', (20, 1))
row = c.execute('SELECT * FROM tasks').fetchone()
print(row)
conn.commit()
conn.close()
```

```
(1, 'My first task', 20)
```

Eliminación de los datos

[17]:
```
conn = pyodbc.connect("Driver=SQLite3;Database=./files/database.db")
c = conn.cursor()
c.execute('DELETE FROM tasks WHERE id = ?', (1,))
row = c.execute('SELECT * FROM tasks').fetchone()
print(row)
conn.commit()
conn.close()
```

```
(2, 'My second task', 5)
```

0.4 Pandas

La librería pandas posee el método `read_sql()` para realizar la conexion a la base de datos usando comandos SQL.

Conexión a la base de datos

[28]:
```
import pandas as pd
import sqlite3

# Establish a connection to an SQLite database
conn = sqlite3.connect('./files/database.db')

df = pd.read_sql("SELECT * FROM tasks", conn)
conn.close()
```

[29]:
```
df
```

[29]:

	id	name	priority
0	2	My second task	5
1	3	My third task	10

Apartir de este punto podemos trabajar con un dataframe usando todas las funcionalidades de Pandas.

Una vez que realizamos los cambios podemos volcarlos de nuevo a la base de datos o agregar nuevos datos, para ello pandas posee del método `to_sql()`, donde:

- name: definimos la tabla donde queremos añadir los datos.
- con: definimos la conexión a la base de datos.
- if_exists: podemos definir como añadir los datos:
  - fail: Genera un ValueError.
  - replace: Borra la tabla antes de insertar nuevos valores.
  - append: Inserta nuevos valores en la tabla existente.
- index: por defecto es True, si no queremos que nos añada el index a la tabla pondremos False
- chunksize (opcional): Especifica el número de filas de cada lote que se escribirán simultáneamente. Por defecto, todas las filas se escribirán a la vez.
- dtype (opcional): Especifica el tipo de dato de las columnas. Si se utiliza un diccionario, las claves deben ser los nombres de las columnas y los valores deben ser los tipos de SQLAlchemy o cadenas para el modo heredado de SQLite3. Si se proporciona un escalar, se aplicará a todas las columnas.

[37]:
```
df_2 = pd.DataFrame({'id': [4], 'name': ['My fourth task'], 'priority': [9]})
df_2
```

[37]:

	id	name	priority
0	4	My fourth task	9

[38]:
```
# Establish a connection to an SQLite database
conn = sqlite3.connect('./files/database.db')

df_2.to_sql(name='tasks', con=conn, if_exists='append', index=False)
```

[38]:
```
1
```

Creado por:

Isabel Maniega