Acceso a Bases de Datos (Teórico)

tiiMiit by

Introducción

Bienvenidos al módulo de Acceso a Bases de Datos

En este módulo abordaremos los de acceso a bases de datos a través del concepto de ORM (Object Relational Mapper) utilizando SQLAlchemy en Python.

Veremos las ventajas de ORM a través de su mapeo de tablas de bases de datos relacionales en clases de la programación orientada a objetos.

Posteriormente veremos qué es SQLAlchemy, sus características principales, sus componentes fundamentales como Engine, Session, Dialectos, Conexiones, etc.

Finalmente una vez terminada la parte teórica iremos a la práctica a trabajar con SQLAlchemy sobre PostgreSQL.

Una vez finalizado este módulo serás capaz de:

- Comprender la importancia de un ORM para trabajar en Python con datos de una base de datos relacional.
- Utilizar SQLAlchemy para conectarse a bases de datos relacionales como PostgreSQL.
- Usar la interfaz de mapeos DBAPI para operar con las bases de datos.
- Conocer los componentes fundamentales de SQLAlchemy necesarios para utilizarlo Engine, Session, Dialectos y Conexiones.

Tema 1. Object Relational Mapping (ORM)

Objetivos

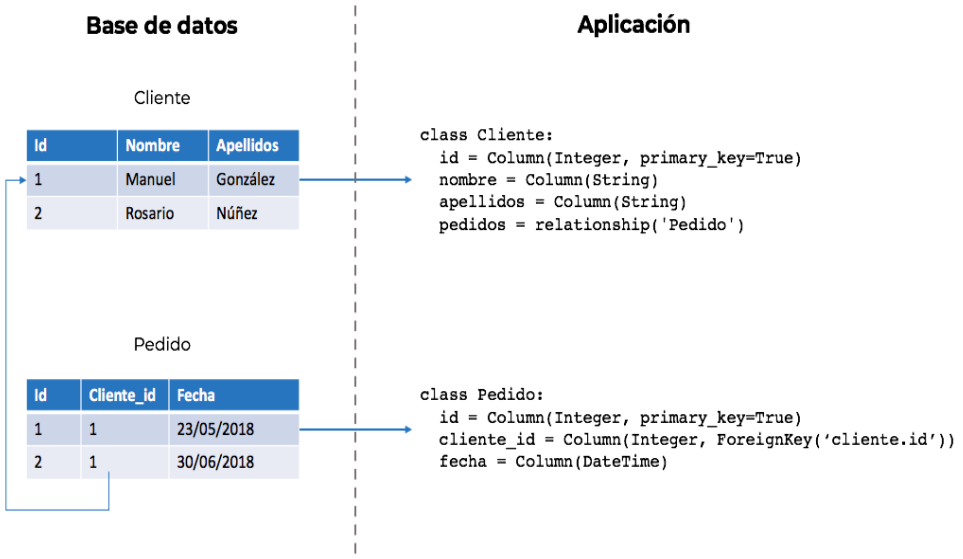
El objetivo de este primer tema es comprender el concepto de Object Relational Mapping (ORM) que nos permitirá entender la manera en que podemos trasladar las tablas de una base de datos relacional a una programación orientada a objetos.

¿Qué es ORM?

Un ORM (de sus siglas en inglés, Object Relational Mapper), no es más que una utilidad o librería que permite manipular las tablas de una base de datos como si fueran objetos de nuestro programa.

Lo más habitual es que una tabla se corresponda con una clase, cada fila de una tabla con un objeto (o instancia de una clase), las columnas de una tabla con los atributos de una clase y las claves ajenas (o Foreign Keys) con relaciones entre clases (definidas también a partir de atributos).

La siguiente imagen muestra la correspondencia entre los elementos de una base de datos y las clases y objetos de un programa:



Ventajas de ORM

A continuación vamos a mencionar las principales ventajas de usar un ORM, entre ellas podemos destacar que:

- Permite acceder a las tablas y filas de una base de datos como clases y objetos.
- En la mayoría de ocasiones no es necesario usar el lenguaje SQL ya que el ORM se encarga de hacer las traducciones oportunas.
- Genera una independencia del tipo base de datos relacional que se esté utilizando, es decir que es posible cambiar de motor de base de datos modificando muy poco código en la aplicación.
- Incrementa la productividad del desarrollador.

Estas ventajas mencionadas irán tomando dimensión al abordar los siguientes temas teóricos y prácticos.

Tema 2. SQLAlchemy

Objetivos

Al finalizar este tema comprenderemos qué es SQLAlchemy y su función para conectarnos a distintas bases de datos mediante un ORM desde Python.

SQLAlchemy

Muchas aplicaciones manipulan información que está almacenada en una base de datos. En Python existen múltiples opciones para acceder y trabajar con una base de datos. Puedes usar directamente conectores que implementan la interfaz de comunicación con las bases de datos más conocidas, como PostgreSQL, MySQL, Oracle, Mongo, etc. O bien, puedes usar SQLAlchemy.

Tal y como indican en la propia web del proyecto, SQLAlchemy es un kit de herramientas SQL para Python y un ORM (Object Relational Mapper) que brinda a los desarrolladores de aplicaciones toda la potencia y flexibilidad de SQL.

SQLAlchemy es una librería para Python que facilita el acceso a una base de datos relacional, así como las operaciones a realizar sobre la misma. Es independiente del motor de base de datos a utilizar, es decir, en principio, es compatible con la mayoría de bases de datos relacionales conocidas: PostgreSQL, MySQL, Oracle, Microsoft SQL Server, Sqlite.

Aunque se puede usar SQLAlchemy utilizando consultas en lenguaje SQL nativo, la principal ventaja de trabajar con esta librería se consigue haciendo uso de su ORM. El ORM de SQLAlchemy mapea tablas a clases Python y convierte automáticamente llamadas a funciones dentro de estas clases a sentencias SQL.

Además, SQLAlchemy implementa múltiples patrones de diseño que permiten desarrollar aplicaciones rápidamente y te abstrae de ciertas tareas, como manejar el pool de conexiones a la base de datos.

Características SQLAlchemy

Las principales características de SQLAlchemy incluyen:

- Un ORM de potencia industrial, construido desde el núcleo en el mapa de identidad, la unidad de trabajo y los patrones del mapeador de datos. Estos patrones permiten la persistencia transparente de objetos utilizando un sistema de configuración declarativo. Los modelos de dominio se pueden construir y manipular de forma natural, y los cambios se sincronizan con la transacción actual automáticamente.
- Un sistema de consulta orientado a la relación, que expone explícitamente toda la gama de capacidades de SQL, incluidas combinaciones, subconsultas, correlaciones y casi todo lo demás, en términos del modelo de objetos. Las consultas de escritura con el ORM utilizan las mismas técnicas de composición relacional que utiliza al escribir SQL. Si bien puede caer en SQL literal en cualquier momento, virtualmente nunca es necesario.
- Un sistema completo y flexible de carga impaciente para colecciones y objetos relacionados. Las colecciones se almacenan en caché dentro de una sesión y se pueden cargar en un acceso individual, todo de una vez mediante uniones, o por consulta por colección en todo el conjunto de resultados.
- Un sistema de construcción Core SQL y una capa de interacción DBAPI. SQLAlchemy Core es independiente del ORM y es una capa de abstracción de base de datos completa por derecho propio, e incluye un lenguaje de expresión SQL basado en Python extensible, metadatos de esquema, agrupación de conexiones, coacción de tipos y tipos personalizados.
- Se supone que todas las restricciones de clave primaria y externa son compuestas y naturales. Por supuesto, las claves primarias de enteros sustitutos siguen siendo la norma, pero SQLAlchemy nunca asume ni codifica los códigos de este modelo.
- Base de datos de introspección y generación. Los esquemas de la base de datos se pueden «reflejar» en un solo paso en las estructuras de Python que representan los metadatos de la base de datos; esas mismas estructuras pueden generar declaraciones CREATE de inmediato, todas dentro del Core, independientemente del ORM.

¿Cómo funciona SQLAlchemy?

SQLAlchemy proporciona una interfaz única para comunicarse con los diferentes drivers de bases de datos Python que implementan el estándar Python DBAPI.

Este estándar, especifica cómo las librerías Python que se integran con las bases de datos deben exponer sus interfaces. Por tanto, al usar SQLAlchemy no se interactúa directamente con dicha API, sino con la interfaz que precisamente proporciona SQLAlchemy. Esto es lo que permite cambiar el motor de base de datos de una aplicación sin modificar apenas el código que interactúa con los datos.

En definitiva, al usar SQLAlchemy es necesario instalar también un driver que implemente la interfaz DBAPI para la base de datos que vayas a utilizar.

Ejemplos de estos drivers a tener en cuenta son:

- psycopg para PostgreSQL
- mysql-connector para MySQL
- cx_Oracle para Oracle

Tema 3. Componentes de SQLAlchemy

Objetivos

El objetivo del siguiente tema es detallar los diversos componentes que son necesarios para poner en marcha SQLAlchemy dentro de Python, entre los cuales veremos Engine, Conexiones, Dialectos, Sesiones, Modelos, Interfaz y Mapeos DBAPI y Database URL

Engine

Lo primero que hay que hacer para trabajar con SQLAlchemy es crear un engine. El engine es el punto de entrada a la base de datos, es decir, el que permite a SQLAlchemy comunicarse con esta.

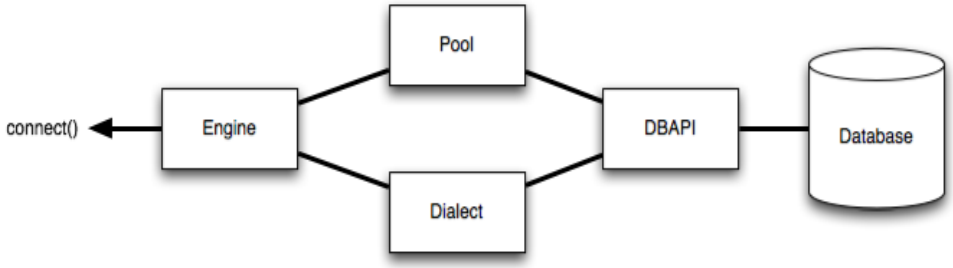
El motor se usa principalmente para manejar dos elementos: los pools de conexiones y el dialecto a utilizar.

Para crear un engine se debe añadir un nuevo módulo Python llamado db.py con el siguiente código:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///productos.sqlite')
```

Como se puede observar, a la función create_engine() se le pasa la cadena de conexión a la base de datos. En este caso, la cadena de conexión a la base de datos Sqlite es 'sqlite:///productos.sqlite'.

Crear el engine no hace que la aplicación se conecte a la base de datos inmediatamente, este hecho se pospone para cuando sea necesario.



Conexiones

SQLAlchemy utiliza el patrón Pool de objetos para manejar las conexiones a la base de datos. Esto quiere decir que cuando se usa una conexión a la base de datos, esta ya está creada previamente y es reutilizada por el programa.

La principal ventaja de este patrón es que mejora el rendimiento de la aplicación, dado que abrir y gestionar una conexión de base de datos es una operación costosa y que consume muchos recursos.

Al crear un engine con la función create_engine(), se genera un pool QueuePool que viene configurado como un pool de 5 conexiones como máximo. Esto se puede modificar en la configuración de SQLAlchemy.

Dialectos

A pesar de que el lenguaje SQL es universal, cada motor de base de datos introduce ciertas variaciones propietarias sobre dicho lenguaje. A esto se le conoce como dialecto.

Una de las ventajas de usar SQLAlchemy es que, en principio, no te tienes que preocupar del dialecto a utilizar. El engine configura el dialecto por ti y se encarga de hacer las traducciones necesarias a código SQL. Esta es una de las razones por las que puedes cambiar el motor de base de datos realizando muy pocos cambios en tu código.

Sesiones

Una vez creado el engine, lo siguiente que debes hacer para trabajar con SQLAlchemy es crear una sesión. Una sesión viene a ser como

una transacción, es decir, un conjunto de operaciones de base de datos que, bien se ejecutan todas de forma atómica, bien no se ejecuta ninguna (si ocurre un fallo en alguna de las operaciones).

Desde el punto de vista de SQLAlchemy, una sesión registra una lista de objetos creados, modificados o eliminados dentro de una misma transacción, de manera que, cuando se confirma la transacción, se reflejan en base de datos todas la operaciones involucradas (o ninguna si ocurre cualquier error).

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///keywords.sqlite')
Session = sessionmaker(bind=engine)
session = Session()
```

Para crear una sesión se utiliza el método factoría sessionmaker() asociado a un engine. Después de crear la factoría, objeto Session, hay que hacer llamadas a la misma para obtener las sesiones, objeto session.

Modelos

Llegados a este punto, ya lo tenemos casi todo listo para interactuar con el ORM. Ahora veamos donde realmente ocurre lo más interesante: los modelos.

Los modelos son las clases que representan las tablas de base de datos. En el ejemplo tenemos la tabla producto, por tanto, dado que estamos usando un ORM, tenemos que crear el modelo (o clase) equivalente a la misma.

Para que se pueda realizar el mapeo de forma automática de una clase a una tabla, y viceversa, vamos a utilizar una clase base en los modelos que implementa toda esta lógica.

Tomaremos como referencia la siguiente tabla de base de datos para introducir ciertos conceptos a continuación

Producto

Id	Nombre	Precio
1	Arroz	1,25
2	Agua	0,30

En el fichero db.py veremos lo siguiente

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///productos.sqlite')
Session = sessionmaker(bind=engine)
session = Session()

Base = declarative_base()
```

Al final del mismo se crea una clase llamada Base con el método declarative_base(). Esta clase será de la que hereden todos los modelos y tiene la capacidad de realizar el mapeo correspondiente a partir de la metainformación (atributos de clase, nombre de la clase, etc.) que encuentre, precisamente, en cada uno de los modelos.

Por tanto, lo siguiente que se debe hacer es crear el modelo Producto creando un nuevo fichero en el directorio productos llamado models.py y añadiendo el código que se indica a continuación:

```python
import db

from sqlalchemy import Column, Integer, String, Float


class Producto(db.Base):
    __tablename__ = 'producto'

    id = Column(Integer, primary_key=True)
    nombre = Column(String, nullable=False)
    precio = Column(Float)

    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def __repr__(self):
        return f'Producto({self.nombre}, {self.precio})'

    def __str__(self):
        return self.nombre
```

Interfaz DBAPI

Python, ofrece el acceso a bases de datos estandarizado por la especificación Database API (DB-API)

DBAPI es un conjunto de clases y funciones comunes, estandarizadas, similares para los distintos motores de bases de datos o wrappers alrededor de estos, escritos en Python. Se desarrolla con la finalidad de lograr la consistencia entre todos estos módulos, y ampliar las posibilidades de crear código portable entre las distintas bases de datos.

Gracias a esto, se puede acceder a cualquier base de datos utilizando la misma interfaz,es decir, el mismo código se podría llegar a usar para cualquier base de datos, tomando siempre los recaudos necesarios (lenguaje SQL estándar, estilo de parámetros soportado, etc.)

Por ello, el manejo de bases de datos en Python siempre sigue estos pasos:

- Importar el conector.

- Conectarse a la base de datos (función connect).
- Abrir un Cursor (método cursor de la conexión).
- Ejecutar una consulta (método execute del cursor).
- Obtener los datos (método fetch o iterar sobre el cursor).
- Cerrar el cursor (método close del cursor).

Mapeos DBAPI

La clase Producto del código anterior representa la tabla producto que vimos al comienzo.

Para que se pueda realizar el mapeo automático clase-tabla, la clase hereda de la clase Base que creamos en la sección anterior y que se encuentra en el módulo db.py. Además, hay que especificar el nombre de la tabla a través del atributo de clase __tablename__.

Por otro lado, cada una de las columnas de la tabla tienen su correspondiente representación en la clase a través de atributos de tipo Column. En este caso concreto, los atributos son los siguientes:id, nombre y precio.

Como se observa, SQLAlchemy define distintos tipos de datos para las columnas (Integer, String o Numeric, entre otros). En función del dialecto seleccionado, estos tipos se mapearán al tipo correcto de la base de datos utilizada.

Por último, y no menos importante, es necesario que al menos un atributo de la clase se especifique como primary_key. En el ejemplo es el atributo id. Este será el atributo que representa a la clave primaria de la tabla.

En la mayoría de motores de bases de datos, al especificar una columna de tipo Integer como primary_key, se generará una columna de tipo entero con valores que se incrementan de manera automática. Además, al crear un objeto no es necesario indicar el valor de esta columna ya que lo establecerá la base de datos cuando se confirmen los cambios.

Database URL

La función create_engine() produce un objeto Engine basado en una URL.

El formato de la URL generalmente sigue el RFC-1738, con algunas excepciones, incluido el hecho de que se aceptan guiones bajos, no guiones ni puntos, dentro de la parte del "esquema".

Las URL generalmente incluyen campos de nombre de usuario, contraseña, nombre de host, nombre de la base de datos, así como argumentos de palabras clave opcionales para una configuración adicional.

En algunos casos, se acepta una ruta de archivo y, en otros, un "nombre de fuente de datos" reemplaza las partes de "host" y "base de datos".

La forma típica de una URL de base de datos es:

dialect+driver://username:password@host:port/database

Los nombres de dialectos incluyen el nombre de identificación del dialecto SQLAlchemy, un nombre como sqlite, mysql, postgresql, oracle o mssql.

El nombre del driver es el nombre de la DBAPI que se usará para conectarse a la base de datos. Si no se especifica, se importará un DBAPI "predeterminado" si está disponible; este valor predeterminado suele ser el controlador más conocido disponible para ese backend.

Ejemplos

A continuación se muestran ejemplos de estilos de conexión comunes.

PostgreSQL

El dialecto de PostgreSQL usa psycopg2 como DBAPI predeterminado. Otras DBAPI de PostgreSQL incluyen pg8000 y asyncpg:

```
# default
engine = create_engine("postgresql://scott:tiger@localhost/mydatabase")

# psycopg2
engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/mydatabase")

# pg8000
engine = create_engine("postgresql+pg8000://scott:tiger@localhost/mydatabase")
```

MySQL

El dialecto de MySQL usa mysqlclient como DBAPI predeterminado. Hay otras DBAPI de MySQL disponibles, incluido PyMySQL:

```
# default
engine = create_engine("mysql://scott:tiger@localhost/foo")

# mysqlclient (a maintained fork of MySQL-Python)
engine = create_engine("mysql+mysqldb://scott:tiger@localhost/foo")

# PyMySQL
engine = create_engine("mysql+pymysql://scott:tiger@localhost/foo")
```

Oracle

El dialecto de Oracle usa cx_oracle como DBAPI predeterminado:

```
engine = create_engine("oracle://scott:tiger@127.0.0.1:1521/sidname")

engine = create_engine("oracle+cx_oracle://scott:tiger@tnsname")
```

Servidor SQL de Microsoft

El dialecto de SQL Server usa pyodbc como DBAPI predeterminado. pymssql también está disponible:

```
# pyodbc
engine = create_engine("mssql+pyodbc://scott:tiger@mydsn")

# pymssql
engine = create_engine("mssql+pymssql://scott:tiger@hostname:port/dbname")
```

SQLite

SQLite se conecta a bases de datos basadas en archivos, utilizando el módulo integrado de Python sqlite3 de forma predeterminada.

Como SQLite se conecta a archivos locales, el formato de URL es ligeramente diferente. La porción de "archivo" de la URL es el nombre de archivo de la base de datos. Para una ruta de archivo relativa, esto requiere tres barras:

```
# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine("sqlite:///foo.db")
```

Cierre

En este módulo hemos introducido conceptos sobre el acceso a bases de datos a través del concepto de ORM (Object Relational Mapper) utilizando SQLAlchemy en Python como una librería que nos facilita el trabajo de conexiones a diferentes fuentes de bases de datos.

Vimos las ventajas de ORM a través de su mapeo de tablas de bases de datos relacionales y luego qué es SQLAlchemy, sus características principales y sus componentes fundamentales.

Ahora que ya se ha terminado la parte teórica daremos paso a la práctica a trabajar con SQLAlchemy sobre PostgreSQL.

Referencias

Python - SQLAlchemy
https://j2logo.com/python/sqlalchemy-tutorial-de-python-sqlalchemy-guia-de-inicio/

Librería - SQLAlchemy
https://entrenamiento-frameworks-web-python.readthedocs.io/es/latest/leccion2/sqlalchemy.html


tiimiit
by