

Acceso a Bases de datos con SQLAlchemy

Curso de Data y Desarrollo en Python
Arturo Bernal Mayordomo - 2023



Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro



GOBIERNO
DE ESPAÑA

MINISTERIO
DE INDUSTRIA, COMERCIO
Y TURISMO



Escuela de
organización
industrial

¿Qué es SQLAlchemy?

- SQLAlchemy es una librería de acceso a bases de datos para Python
- Es un ORM (Object-Relational Mapper). ¿Qué es un ORM?:
 - Permite abstraerte en casi todos los escenarios de las consultas SQL, usando solo objetos y métodos de la librería
 - Trabaja con objetos del lenguaje para representar los datos (en lugar de tuplas por ejemplo)
 - Se abstrae del tipo de base de datos. Las operaciones son las mismas utilicemos SQLite, MariaDB, PostgreSQL, Oracle, ...

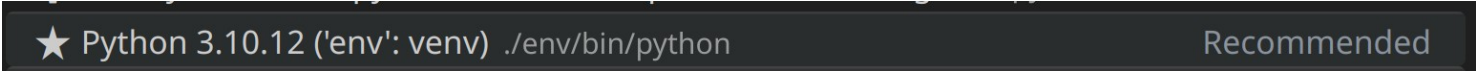
Instalar SQLAlchemy

- Primero activamos un entorno virtual en el proyecto actual para no instalar la librería de forma global. Debe ejecutarse en el directorio donde tengamos la app actual
 - Windows: `py -m venv env`
 - Linux/Mac: `python3 -m venv env`
- A continuación lo activamos:
 - `Ctrl+shift+P` → Python select interpreter → seleccionamos la versión ('env': venv)
- Instalamos el paquete: **`pip install sqlalchemy`**



```
>python select interpre
```

Python: Select Interpreter



★ Python 3.10.12 ('env': venv) ./env/bin/python

Recommended

Clases del Modelo

- Se definen como las DataClass pero con alguna diferencia. Internamente SQLAlchemy las transforma en dataclasses
 - No necesitan el decorador `@dataclass` explícitamente
 - Tenemos que crear una clase que herede de **DeclarativeBase** (la llamaremos por ejemplo **BaseModel**). Puede estar vacía. Las clases del modelo deben heredar de dicha clase
 - Los atributos se definen con anotaciones de tipos (deben ser del tipo **Mapped[tipo]** → `Mapped[str]`, `Mapped[int]`, etc.)
 - Los valores serán llamadas a la función **mapped_column**, donde le pasamos características como si es clave primaria, el tipo de dato, la longitud, etc.
 - Deben incluir el nombre de la tabla a la que hacen referencia en un atributo de clase llamado **__tablename__**

Ejemplo clases del modelo

```
from sqlalchemy import String, Numeric
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column

class BaseModel(DeclarativeBase): # Obligatoria crear y que herede así
    pass

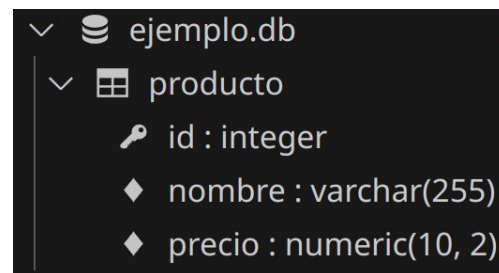
class Producto(BaseModel):
    __tablename__ = 'producto' # Tabla donde se realizan las consultas
    # Aunque SQLite no tenga limitaciones de tamaño en campos, etc., SQLAlchemy los impone
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True) # Clave primaria
    nombre: Mapped[str] = mapped_column(String(255)) # equivale a VARCHAR(255) en MySQL
    precio: Mapped[float] = mapped_column(Numeric(10,2)) # equivale a tipo Decimal en MySQL

    def __repr__(self) -> str:
        return f"{self.id} - {self.nombre} ({self.precio: .2f})"
```

Conexión a la base de datos

- Para conectar a la base de datos usamos la función `create_engine` y le pasamos la cadena de conexión.
 - En SQLite la cadena de conexión es algo como “`sqlite:///datos.bd`”
- A continuación (si no las tenemos) creamos las tablas en la base de datos a partir de las clases del modelo definidas → `BaseModel.metadata.create_all(engine)`
 - Con el parámetro extra **`echo=True`** en `create_engine`, activamos la depuración y mostrará todas las consultas generadas

```
engine = create_engine("sqlite:///ejemplo.db")
BaseModel.metadata.create_all(engine) # Creamos
las tablas en la base de datos
```



Insertar registros

- Para realizar consultas se tiene que abrir una sesión (**Session**). Esto se puede hacer con el bloque **with** para que se cierre automáticamente al acabar.
 - Creamos objetos de las clases del modelo y los añadimos con **add** (un objeto), o **add_all** (lista de objetos)

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, Session

# Se usan parámetros por nombre en el constructor
silla = Producto(nombre = "Silla", precio = 25.95)
mesa = Producto(nombre = "Mesa", precio = 90)
estanteria = Producto(nombre = "Estantería", precio = 67.5)

with Session(engine) as session:
    session.add(silla) # Añadir un producto
    session.add_all([mesa, estanteria]) # Añadir varios a la vez
    session.commit() # Se hacen las insert generando las claves primarias
```

Consultas Select

- Para ello se utiliza la función **select(ClaseModelo)**. Además, se puede concatenar una llamada al método **where** para establecer condiciones
- Los resultados se obtienen generalmente con **scalars().all()** → Lista de objetos, **scalars().first()** → Primer objeto, **scalars().one()** → Único objeto (error si la consulta devuelve más filas o ninguna), **scalars().one_or_none()** → Si el resultado es vacío devuelve None
 - También se puede usar **session().get(ClaseModelo, valor_clave_primaria)** para obtener un único registro como objeto

```
with Session(engine) as session:
    st = select(Producto)
    print(st) # SELECT producto.id, producto.nombre, producto.precio FROM producto
    productos = session.execute(st).scalars().all() # Lista de productos
    print(productos)

    st2 = select(Producto).where(Producto.id == 1) # Condición Where
    producto = session.execute(st2).scalars().one() # Único objeto
    print(producto)
```


Modificar registros

- Para modificar registros (UPDATE), basta con modificar objetos obtenidos **dentro de la misma sesión**, y posteriormente hacer un commit
 - Se pueden ver los cambios a registrar antes del commit imprimiendo `session.dirty`
 - Si el objeto no ha sido obtenido en la sesión actual (bloque with) se puede sincronizar con **`session.merge(objeto)`** primero. A partir de ahí los cambios estarán gestionados en la sesión actual

```
with Session(engine) as session:  
    producto = session.get(Producto, 2)  
    producto.precio += 100  
    session.commit() # Se genera el update
```

Borrar un registro

- Se puede borrar un objeto obtenido en la sesión actual llamando a `session.delete(id)`

```
with Session(engine) as session:  
    producto = session.get(Producto, 2)  
    session.delete(2)  
    session.commit()
```

Gestión de relaciones

- La característica principal de las bases de datos relacionales son las relaciones entre tablas
 - Esto se gestiona por medio de las claves ajenas. Valores en la columna de una tabla que se relacionan con la clave primaria de otra tabla
 - Ej: La tabla **jugador** puede tener una columna llamada equipo_id que sea una clave ajena que apunta a la id de la tabla **equipo**
 - Esto implica que un jugador debe tener asociada una id de equipo que exista como clave primaria en otra tabla. Si no, se poduce un fallo de integridad, no dejando insertar dicho jugador
 - Esto sería la clásica relación 1 a muchos (1-N)
 - Un jugador se relaciona con un equipo (clave ajena)
 - Un equipo puede tener muchos jugadores

Ejemplo de relación 1-N (Producto-Categoría)

```
from sqlalchemy import String, Numeric, create_engine, select, ForeignKey
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, Session, relationship
from typing import List

class Categoria(BaseModel):
    __tablename__ = 'categoria'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    nombre: Mapped[str] = mapped_column(String(255))
    productos: Mapped[List["Producto"]] = relationship(back_populates="categoria")

class Producto(BaseModel):
    __tablename__ = 'producto'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    nombre: Mapped[str] = mapped_column(String(255))
    precio: Mapped[float] = mapped_column(Numeric(10,2))
    categoria_id: Mapped[int] = mapped_column(ForeignKey("categoria.id"))
    categoria: Mapped["Categoria"] = relationship(back_populates="productos")
```

Como crear la relación

- Primero mapeamos la clave ajena (categoria_id en este caso)
 - La columna será del tipo **ForeignKey** y dentro establecemos el campo de la tabla a la que apunta (clave primaria) → “categoria.id”
- Después mapeamos las relaciones. Pueden ser en una dirección (producto tiene categoría, o categoría tiene lista de productos), o bidireccional como en este caso.
 - El campo **back_populates**, significa que automáticamente cuando asociamos un producto a una categoría, la categoría también se asocia al producto, y viceversa. Tiene que apuntar al nombre del atributo de la otra clase que marca la relación (**relationship**)

- Las relaciones permiten cosas como:
 - Añadir una categoría con sus productos en una sola operación
 - Borrar una categoría con todos sus productos
 - Modificar los productos asociados a una categoría metiéndolos y sacándolos de su lista
 - Obtener una categoría y poder navegar por sus productos y viceversa

```
c = Categoria(nombre = "Categoría 1")
c.productos.extend([ # Relacionamos nuevos productos con la categoría
    Producto(nombre = "Producto 1", precio = 4.5), # No hace falta valor a la clave ajena
    Producto(nombre = "Producto 2", precio = 12),
    Producto(nombre = "Producto 3", precio = 6.75),
])

session.add(c) # Añadimos categoría con sus productos
session.commit()
print(c) # 1 - Categoría 1
print(c.productos) # [1 - Producto 1 ( 4.50), 2 - Producto 2 ( 12.00), 3 - Producto 3 ( 6.75)]
```

id	nombre
1	Categoría 1

id	nombre	precio	categoria_id
1	Producto 1	4.5	1
2	Producto 2	12	1
3	Producto 3	6.75	1