

Python / Guía 11

Bases de datos   
con SQLModel

## 

## 

## OBJETIVOS DE LA GUÍA

En esta guía aprenderemos a:

* Usar el moderno ORM: SQLModel, basado en   
  Pydantic y SQLAlchemy
* Crear tablas y columnas
* Crear índices
* Crear filas, modificarlas, consultarlas o eliminarlas
* Consultar con filtros
* Relacionar tablas
* Crear atributos de relación (propio de SQLModel)

**No dejes de ir escribiendo el código y probando, según se te va a ir indicando, porque al hacerlo, no solo te permitirá entender esta guía, sino que el mismo hecho de escribir, aumentará tu experiencia como desarrollador.**

# ¿Por qué SQLModel?

SQLModel es una librería para interactuar con bases de datos SQL desde código Python, con objetos Python. Está diseñado para ser intuitivo, fácil de usar, altamente compatible y robusto. Se basa en anotaciones de tipo Python y funciona con Pydantic y SQLAlchemy, librerías que son el motor de SQLModel.

## PYDANTIC

Pydantic es una librería que permite la validación de datos mediante anotaciones de tipo Python. Aplica sugerencias de tipo en tiempo de ejecución y proporciona errores fáciles de usar cuando los datos no son válidos.

## SQLALCHEMY

SQLAlchemy es uno de los ORM más usados en Python, con muchos años entre los desarrolladores. Incluye soporte para SQLite, MySQL, PostgreSQL, Oracle, MS SQL, entre otros. ORM significa Object-Relational-Mapping: es una técnica que permite convertir entre los tipos de datos usados en los lenguajes de programación orientados a objetos y los tipos de datos de una base de datos relacional.

## Características de SQLModel

SQLModel es una capa delgada sobre Pydantic y SQLAlchemy, diseñada para ser compatible con ambos. Combina SQLAlchemy y Pydantic y simplifica el código que se escribe, lo que te permite reducir la duplicación de código al mínimo, y así vas obteniendo una eficaz experiencia como desarrollador.

* Es Intuitivo para escribir: da soporte el editor de código
* Menos tiempo de depuración: diseñado para ser fácil de usar y aprender: menor tiempo leyendo documentación.
* Fácil de usar: tiene valores predeterminados que hace el trabajo por debajo, para simplificar todo lo que haces en él.
* Compatible: está diseñado para ser compatible con FastAPI, Pydantic y SQLAlchemy.
* Extensible: tienes todo el poder de SQLAlchemy y Pydantic por debajo.
* Corto: Minimiza la duplicación de código. Una sola anotación de ese tipo hace mucho trabajo. No es necesario duplicar modelos en SQLAlchemy y Pydantic.

## Instalación

Como SQLModel se basa en Pydantic y SQLAlchemy, los requiere. Se instalarán automáticamente al instalar SQLModel.

Crea una nueva carpeta para tu proyecto, crea un entorno virtual, actívalo, y luego ejecuta:

**pip install sqlmodel**

| Hay alternativas a los entornos virtuales que son particularmente útiles para la implementación, como Docker y otros tipos de contenedores. En este caso, el "entorno virtual" no consistirá únicamente en los archivos estándares de Python y los paquetes instalados, sino también en todo el sistema. |
| --- |

En Visual Studio Code, busca la extensión llamada SQLite: servirá para ver nuestra base de datos. Si eres usuario de Linux, antes de instalar la extensión, asegúrate de instalar sqlite3: **sudo apt install sqlite3**

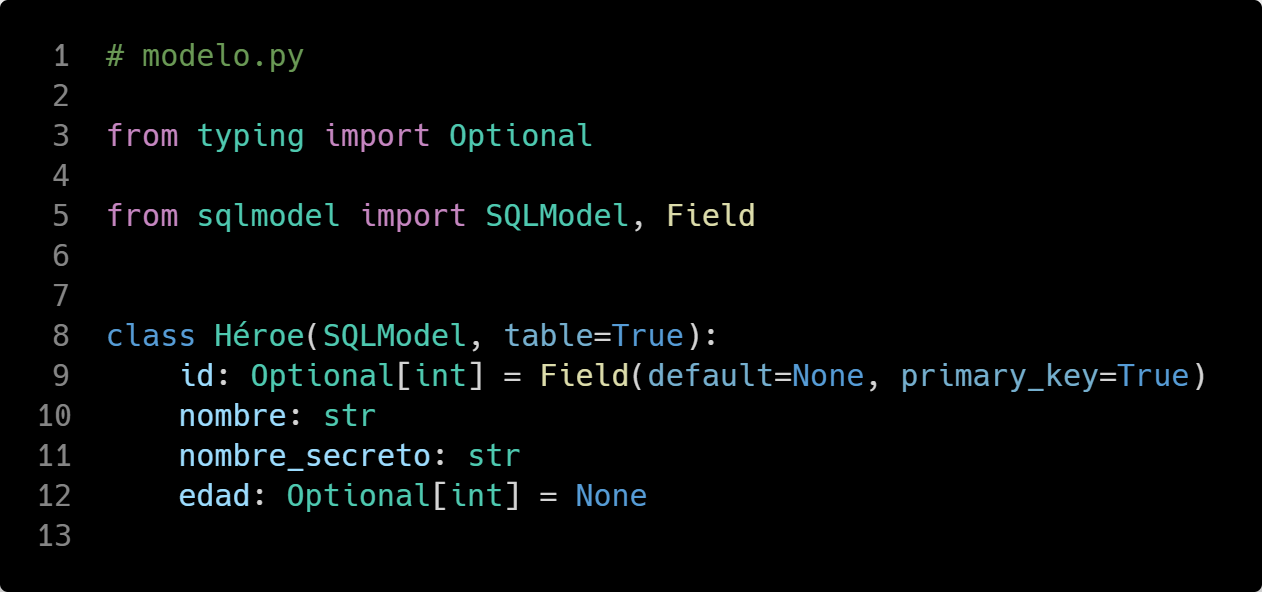
Usaremos la base SQLite para los ejercicios. Python tiene soporte interno para SQLite. Una base de datos SQLite es un único archivo, y no necesita de un servidor de bases de datos externo, por lo que es perfecto para aprender. De hecho, SQLite es perfectamente capaz de manejar aplicaciones bastante grandes, pero en algún momento quizás tendrás que migrar, por ejemplo a PostgreSQL (que también es gratuita), muy apoyada por la comunidad internacional y una de las más rápidas.

# Creación de tablas

Lo primero que debes hacer es crear una clase que represente los datos de la tabla. Una clase que representa los datos de una tabla se denomina comúnmente modelo o "model".

**from sqlmodel import SQLModel, Field**

Para eso, importarás SQLModel y Field, y crearemos una clase que hereda y representa el modelo de tabla para nuestros héroes:



La clase Héroe representa la tabla para nuestros héroes. Y cada instancia que creemos más adelante, representará una fila de la tabla Héroe.

Usamos la configuración para decirle a SQLModel que este es un modelo que representa una tabla: table=True

| También es posible tener modelos sin table=True. Esos serían solo modelos de datos, sin una tabla en la base de datos. Serán útiles más adelante, pero por ahora, seguiremos agregando la configuración table=True |
| --- |

## Campos opcionales, columnas null

Observa que **edad** tiene un tipo **Optional[int]**. De esta forma declaramos que podría ser un valor de tipo **int** o **None**. Establecemos que por defecto es **Non**e. Así le decimos a SQLModel que no es necesario validar datos y que tiene un valor predeterminado. Esta columna es "nullable" (puede ser establecida en **null** dentro de la base de datos) Pydantic lo toma como un campo opcional. Para SQLAlchemy el campo es una columna "nullable".

## Clave primaria (primary key) id

Ahora revisemos el campo **id**. Esta es la clave principal de la tabla. Usamos la función especial **Field** de SQLModel, y establecemos el argumento: **primary\_key=True**.

De esa manera, le decimos a SQLModel que este campo/columna es la clave principal de la tabla. Pero dentro de la base de datos SQL, **siempre es necesario** y no puede ser **null** .

¿Por qué deberíamos declararlo con **Optional**?

El **id** será generado por la base de datos, no por nuestro código, por lo tanto, cada vez que creemos una instancia de esta clase, no establecemos el **id**, y el valor de **id** será **None hasta que lo guardemos en la base de datos**, y entonces sí tendrá un valor.

Para entender mejor, supongamos el siguiente escenario (hacer\_algo es una función imaginativa):

mi\_héroe = Héroe(nombre="Spider-Boy", nombre\_secreto="sb")

hacer\_algo(mi\_héroe.id)

mi\_heroe.id será **None**

Pero imagina que guardamos en la base de datos:

guardar\_en\_mi\_base(mi\_héroe)

hacer\_algo(mi\_héroe.id)

Ahora sí, **mi\_héroe.id** tiene un valor, que es un **int**, dado por la base de datos, no por nuestro código.

Entonces, porque en nuestro código (no en la base de datos) el valor de **id** podría ser **None**, usamos **Optional**. De esta manera, **el editor será capaz de ayudarnos**, por ejemplo, si intentamos acceder al id de un objeto que aún no hemos guardado en la base de datos y aún no hemos guardado.

Usamos **Field(default=None)**, para que, cada vez que usemos este modelo y se haga la validación de datos (la validación la hace Pydantic), acepte un valor de tipo **None** (lo cual sirve al desarrollador) y además un valor tipo **int** (lo usa la base de datos). Por eso, es necesario declarar la clave primaria de esta forma.

## 

## Creación del motor

Ahora necesitamos crear el Motor SQLAlchemy. Es un objeto que maneja la comunicación con la base de datos. Si tienes una base de datos de servidor (por ejemplo, PostgreSQL o MySQL), el **motor** contendrá las **conexiones de red** a esa base de datos.

from sqlmodel import SQLModel, Field, create\_engine

Llama a **create\_engine()** con una URL para usar la base de datos.

sqlite\_nombre = "database.db"  
 sqlite\_url = f"sqlite:///{sqlite\_nombre}"  
 engine = **create\_engine(sqlite\_url**, echo=True)

| Normalmente deberías tener un solo objeto de motor para toda la aplicación y reutilizarlo en todas partes.  Puedes leer más sobre las bases de datos compatibles con SQLAlchemy (y de esa manera compatibles con SQLModel) en la documentación de SQLAlchemy <https://docs.sqlalchemy.org/en/14/core/engines.html> |
| --- |

En este ejemplo, también estamos usando el argumento echo=True. Hará que el motor imprima todas las instrucciones SQL que ejecuta, lo que puede ayudarte a comprender lo que está sucediendo. Es útil para el aprendizaje y la depuración.

## Creación de la base de datos y tabla

La creación del motor no crea el archivo database.db. Para eso debes ejecutar:

SQLModel.metadata.create\_all(engine)

Crea el archivo y crea la tabla en esa base de datos. Ambas en un solo paso.

### SQLModel.metadata

La clase SQLModel tiene un atributo metadata. Es una instancia de la clase MetaData. Cada vez que creas una clase que hereda SQLModel y está configurada table=True , se registra en este objeto MetaData.

### create\_all()

El objeto MetaData en SQLModel.metadata tiene un método create\_all(). Recibe un motor y lo usa para crear la base de datos y todas las tablas registradas en el objeto MetaData.

### Importancia del orden de los metadatos de SQLModel

Por ejemplo, imaginemos que haces esto:

* Creas los modelos en un archivo de Python model.py.
* Creas el objeto del motor en un archivo database.py.
* Creas tu aplicación principal y llamas a SQLModel.metadata.create\_all() en app.py.

Si solo importaste SQLModel e intentaste llamar a SQLModel.metadata.create\_all() desde app.py, las tablas no se crearán:

# Esto no funciona

from sqlmodel import SQLModel

from .database import engine

SQLModel.metadata.create\_all(engine)

No funciona, porque cuando importas solamente a SQLModel, Python no ejecuta todo el código creando las clases que heredan de él (en nuestro ejemplo, la clase Héroe), porque SQLModel.metadata todavía está vacío.

Pero si importas los modelos antes de llamar SQLModel.metadata.create\_all(), funcionará:

from sqlmodel import SQLModel

**from . import model**

from .database import engine

SQLModel.metadata.create\_all(engine)

Funciona porque, al importar los modelos, Python ejecuta todo el código creando las clases heredadas SQLModel, y las registra en SQLModel.metadata.

Como alternativa, puedes importar SQLModel y tus modelos dentro de database.py:

# database.py

from sqlmodel import SQLModel, create\_engine

from . import model

sqlite\_nombre= "database.db"

sqlite\_url = f"sqlite:///{sqlite\_nombre}"

engine = create\_engine(sqlite\_url)

Y luego importar SQLModel desde database.py adentro app.py, y ahí llamar SQLModel.metadata.create\_all():

# app.py

from .database import engine, SQLModel

SQLModel.metadata.create\_all(engine)

### Migraciones

Para este simple ejemplo, usar SQLModel.metadata.create\_all() es suficiente.

Pero para un sistema de producción, querrás usar un sistema para migrar la base de datos. Esto sería útil e importante, por ejemplo, cuando agregues o elimines una columna, agregues una nueva tabla, cambies un tipo, etc.

| La librería **Alembic** es una buena solución para las migraciones. |
| --- |

### TEXT o VARCHAR

Cada base de datos admite algunos tipos de datos particulares , como INTEGER y TEXT. Algunas bases de datos tienen algunos tipos particulares, por ejemplo, PostgreSQL y MySQL usan BOOLEAN para los valores de True y False. SQLite acepta SQL con booleanos, incluso cuando define las columnas de la tabla, pero lo que usa internamente son INTEGER: con 1 representa True, y con 0 representa False.

De la misma manera, hay varios tipos posibles para almacenar cadenas. SQLite usa TEXT. Pero otras bases de datos como PostgreSQL y MySQL usan VARCHAR.

SQLAlchemy genera las instrucciones SQL para crear tablas usando VARCHAR, y luego SQLite las recibe e internamente las convierte a TEXT.

Además de la diferencia entre esos dos tipos de datos, algunas bases de datos como MySQL requieren establecer una longitud máxima para los VARCHAR, por ejemplo, VARCHAR(255) establece la cantidad máxima de caracteres en 255.

SQLModel simplifica todo esto. Independientemente de la base de datos que uses, y sin ninguna configuración adicional, de manera predeterminada, los campos str se interpretan como VARCHAR para la mayoría de las bases de datos y VARCHAR(255) para MySQL. De esta manera, el modelo de datos será compatible con cualquier base de datos popular, sin más esfuerzo que usar la anotación de tipo str.

## 

## Índices

Los índices de base de datos almacenan algunas claves, y registros, de manera tal que hace que sea más fácil y rápido de encontrar el dato buscado.

Imaginemos que tienes muchas filas. Si le dices a la base de datos que obtenga una fila por un "nombre" específico, la base de datos tendrá que escanear todos los registros, verificando uno por uno, lo que sería muy lento.

Pero si la base de datos tuviera un índice para la columna "nombre", se ordenaría de forma alfabética, ya que es una cadena. De esa manera, en lugar de leer miles de registros, la base de datos podría hacer unos pocos pasos, digamos 5 o 10 pasos, y llegar a la fila del índice que tiene.

Hay un costo asociado con los índices. Cuando no tienes un índice y agregas una nueva fila a la tabla, la base de datos tiene que realizar 1 operación para agregar la nueva fila al final de la tabla. Pero si creas un índice para "nombres", ahora la base de datos tiene que proceder a la misma operación para agregar esa fila más 5 o 10 operaciones adicionales en el índice, para encontrar el lugar correcto para el nombre, para luego agregar ese registro de índice allí.

Y si le agregas un índice más llamado "edad", ahora la base de datos tiene que realizar la misma operación para agregar esa fila, más 5 o 10 operaciones más para agregar el índice "edad", más 5 o 10 operaciones para agregarlo en el índice "nombres". Esto significa que ahora, con 2 índices creados, agregar una fila requiere como unas 30 operaciones.

Pero estás negociando el tiempo en el que se tarda en leer los datos por el tiempo en el que se tarda en escribirlos, además, un poco de espacio adicional en la base de datos.

Si tienes consultas que piden datos constantemente de esos campos, entonces sí tiene mucho sentido tener índices para cada uno de ellos. Porque 31 operaciones al crear o actualizar datos (más el espacio del índice) es mucho mejor que las posibles 500 o 1000 operaciones para leer todas las filas para poder compararlas. Pero si nunca tienes consultas en alguna columna, no tiene sentido tener un índice, ya que eso aumentará el costo computacional y de espacio de escribir y actualizar la base de datos.

Crear un índice en SQLModel es tan sencillo como hacer lo siguiente en el modelo:

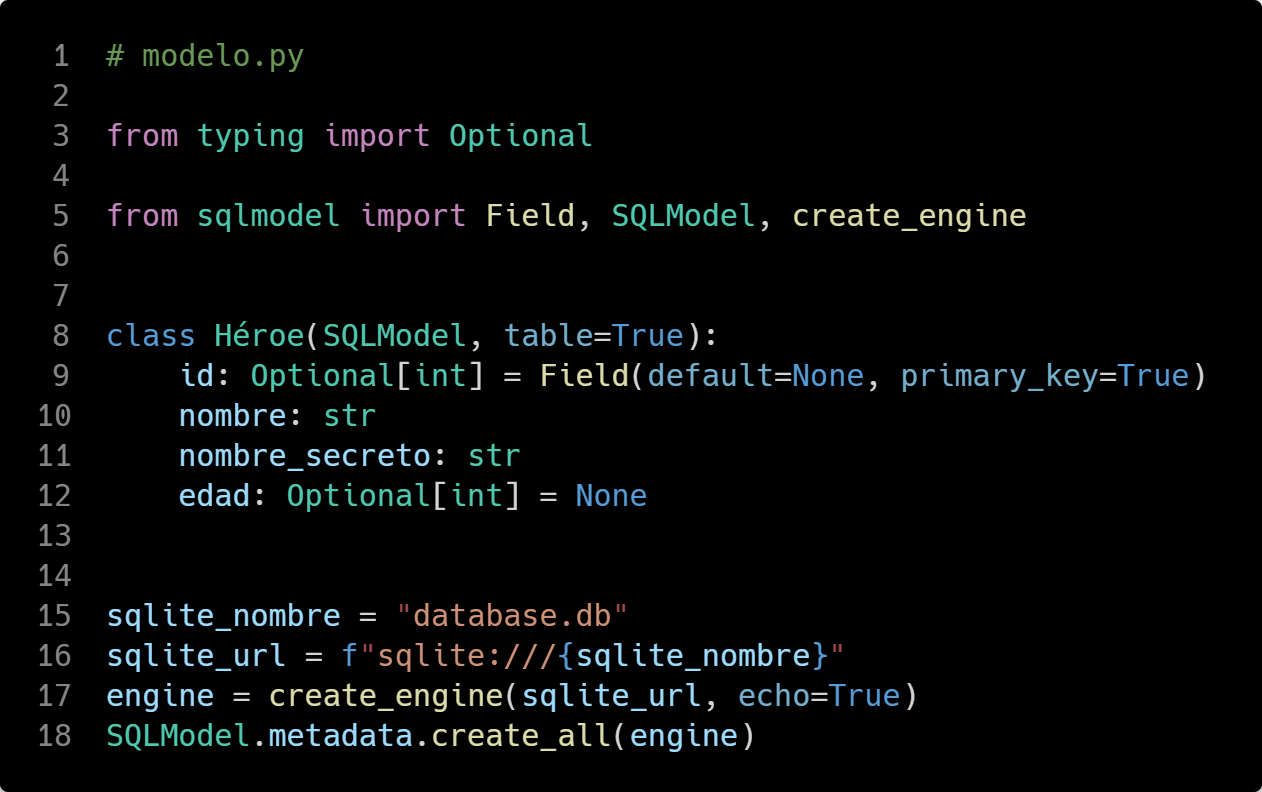
nombre: str = Field(index=True)

RESTRICCIÓN UNIQUE

La restricción UNIQUE se utiliza para garantizar que no se inserten valores duplicados en una columna específica.

nombre: str = Field(index=True, unique=True)

**¡MANOS A LA OBRA!** 



1. **Verifica la base de datos:** Ahora que has creado el archivo y la tabla, en Visual Studio Code haz clic con el botón derecho del mouse sobre el database.db y elige del menú desplegable: "Open Database". Verás que en la barra izquierda llamada "Explorer" o "Explorador", una pestaña llamada "SQLITE EXPLORER". Ahí podrás desplegar el contenido y ver la estructura de la tabla.
2. Reestructura el código, para que sea reutilizable, lo puedas compartir y luego testear. Para eso, crea una función que se llame crear\_db\_y\_tablas(), donde anidarás la instrucción para crear la base de datos y las tablas.
3. Crea if \_\_name\_\_ == "\_\_main\_\_" y adentro anida el llamado a la función recién creada.
4. Ejecuta el archivo modelo.py dos veces.
5. Agrega la restricción unique a la columna nombre y prueba guardar 2 veces el mismo nombre.

## Revisemos lo aprendido hasta aquí

* Conocer las características de SQLModel
* Instalar SQLModel
* Crear tablas
* Comprender el modo de la creación de la clave primaria
* Comprender el funcionamiento esencial de SQLModel respecto al motor
* La importancia del orden de las importaciones
* Crear índices
* Crear la restricción unique para una columna

# Creación de las filas (INSERT)

## Creación de instancias de modelo

Como primer paso, hay que crear los datos en la memoria. Ya hemos creado una clase Héroe que representa la tabla héroe en la base de datos.

Cada instancia que creemos, representará los datos en una fila en la base de datos.

def crear\_héroes():

héroe\_1 = Héroe(nombre="Deadpond", nombre\_secreto="Dive")

héroe\_2 = Héroe(nombre="Spider-Boy", nombre\_secreto="Pedro")

héroe\_3 = Héroe(nombre="Rusty-Man", nombre\_secreto="Tommy", edad=48)

## Creación de una sesión y adición de datos

A diferencia del **motor,** que es uno para toda la aplicación, necesitamos una **sesión** para cada grupo de operaciones que queramos hacer con la base de datos. De hecho, la sesión necesita un motor:

from sqlmodel import SQLModel, Field, create\_engine, Session

Por ejemplo, si tenemos una aplicación web, normalmente tendríamos una sola sesión por solicitud. Realizaríamos el mismo motor en todo el código, en todas partes de la aplicación.

Por cada solicitud que tenemos, crearemos y usaremos una nueva sesión. Y una vez hecha la operación, cerramos la sesión.

SQLModel nos trae la facilidad de usar el administrador de contexto (context manager), es decir, el bloque **with - as**. Recuerda su función con open... Con la sesión se hace lo mismo: se abre la sesión, se ejecuta el bloque, y cuando termina de hacerlo se cierra la sesión de forma automática.

Para agregar instancias a la sesión usamos el método add. Una vez que tengamos listas las operaciones, podemos "comprometer" (como git commit) todos los cambios en un solo paso: session.commit() Esto hace que se escriban los datos en database.db.

**with Session(engine) as session**:

session.**add**(héroe\_1)

session.**add**(héroe\_2)

session.**add**(héroe\_3)

session.**commit**()

**¡MANOS A LA OBRA!** 

1. Elimina el archivo database.db, para que podamos comenzar desde cero.
2. Lleva el bloque with (☝️ ) dentro de la función crear\_héroes.
3. Llama a la función crear\_héroes después de crear la base de datos y las tablas.
4. Verifica el contenido de la base de datos con la extensión SQLITE de Visual Studio Code.
5. Vuelve a ejecutar modelos.py y revisa el contenido de la base de datos.
6. Intenta modificar un héroe sin el argumento nombre\_secreto, ejecuta y comprueba en la base

|  | **Ver Video “**[**Bases de datos parte 1**](https://youtu.be/dbGp6JKNFWg)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Crear instancias de un modelo
* Abrir una sesión con Session(engine)
* Agregar filas con <instancia de Session>.add(<instancia de modelo>)
* Guardar los cambios con <instancia de Session>.commit()

## Id automáticos, none predeterminado y actualización

Decíamos que el campo **id** no puede ser **NULL** en la base de datos porque es una clave principal, y lo declaramos usando **Field(primary\_key=True)**.

Ahora, veremos el tema sobre la sincronización de datos entre la base de datos y el código de Python.

Podemos confirmar lo dicho sobre la clave primaria imprimiendo a nuestros héroes antes de agregarlos a la base de datos

| Escribe el siguiente código inmediatamente después de la creación de las instancias, comenta el bloque with Session, para que no se ejecute, y luego corre el programa. |
| --- |

print("Antes de comitear la sesión:")  
print("Héroe 1:", héroe\_1)  
print("Héroe 2:", héroe\_2)

print("Héroe 3:", héroe\_3)

**¿Héroe 1: id=None?**

| Ahora descomenta el bloque with Session, escribe el siguiente código dentro del bloque with, después de la línea session.commit(), y corre el programa |
| --- |

print("Después de comitear la sesión:")  
print("Héroe 1:", héroe\_1)  
print("Héroe 2:", héroe\_2)  
print("Héroe 3:", héroe\_3)

**¿Héroe 1:?**

Lo que sucede es que SQLModel (en realidad SQLAlchemy) está marcando internamente esos objetos como "vencidos" o "caducados". Se puede explicar esto con un ejemplo:

Supongamos que hemos creado un campo llamado "actualizado\_en" de tipo datetime, que se acaba de guardar en la base de datos a las 12hs12m12s. Si siguiéramos con el objeto "vivo" durante un tiempo e intentamos hacer una operación con el objeto, podríamos obtener inconsistencias graves: el objeto guardó una transacción de dinero a las 12hs, y más tarde, por error, guardó la misma transacción en otra tabla a las 13hs. Creo que serías sospechoso de fraude. En efecto, por eso los objetos se marcan como "vencidos".

## Refrescado de objetos ya comiteados

SQLModel puede traer el valor guardado en la base de datos en el objeto, para que lo puedas utilizar nuevamente teniendo la "última versión" de los datos.

Puedes refrescar de dos formas: de forma implícita o explícita:

### De forma implícita

**Accede a un atributo del objeto**, y esto va a "refrescar" los datos para establecerlos en el atributo del objeto y ponerlos a tu disposición.

| Escribe el siguiente código y ponlo en el bloque with, al final. Ejecuta el programa. |
| --- |

print("Después de comitear la sesión: muestra IDs")  
 print("Héroe 1 id:", héroe\_1.**id**)  
 print("Héroe 2 id:", héroe\_2.**id**)  
 print("Héroe 3 id:", héroe\_3.**id**)  
 print("Después de comitear la sesión: muestra nombres")  
 print("Héroe 1 nombre:", héroe\_1.**nombre**)  
 print("Héroe 2 nombre:", héroe\_2.**nombre**)  
 print("Héroe 3 nombre:", héroe\_3.**nombre**)

¿Héroe 1 nombre: Deadpond? Ahora sí, se refrescó el objeto, de forma implícita.

### De forma explícita

Usando el método refresh() la sesión hace que el motor se comunique con la base de datos para obtener los datos recientes del objeto, luego la sesión coloca los datos en el objeto y lo marca como "nuevo" o "no vencido".

session.**refresh**(héroe\_1)

session.**refresh**(héroe\_2)

session.**refresh**(héroe\_3)

print("Después de refrescar los héroes")

print("Héroe 1:", héroe\_1)

print("Héroe 2:", héroe\_2)

print("Héroe 3:", héroe\_3)

| Esto podría ser útil cuando crees una API web. Una vez que se guardan datos de un objeto en la base de datos, se los devuelve al cliente. |
| --- |

Ahora, como experimento final, también podemos imprimir datos después de cerrar la sesión.

| Escribe el siguiente código fuera del bloque with, pero sin salir de la función crear\_héroes(). Ejecuta el programa. |
| --- |

print("Después de que se cierra la sesión")

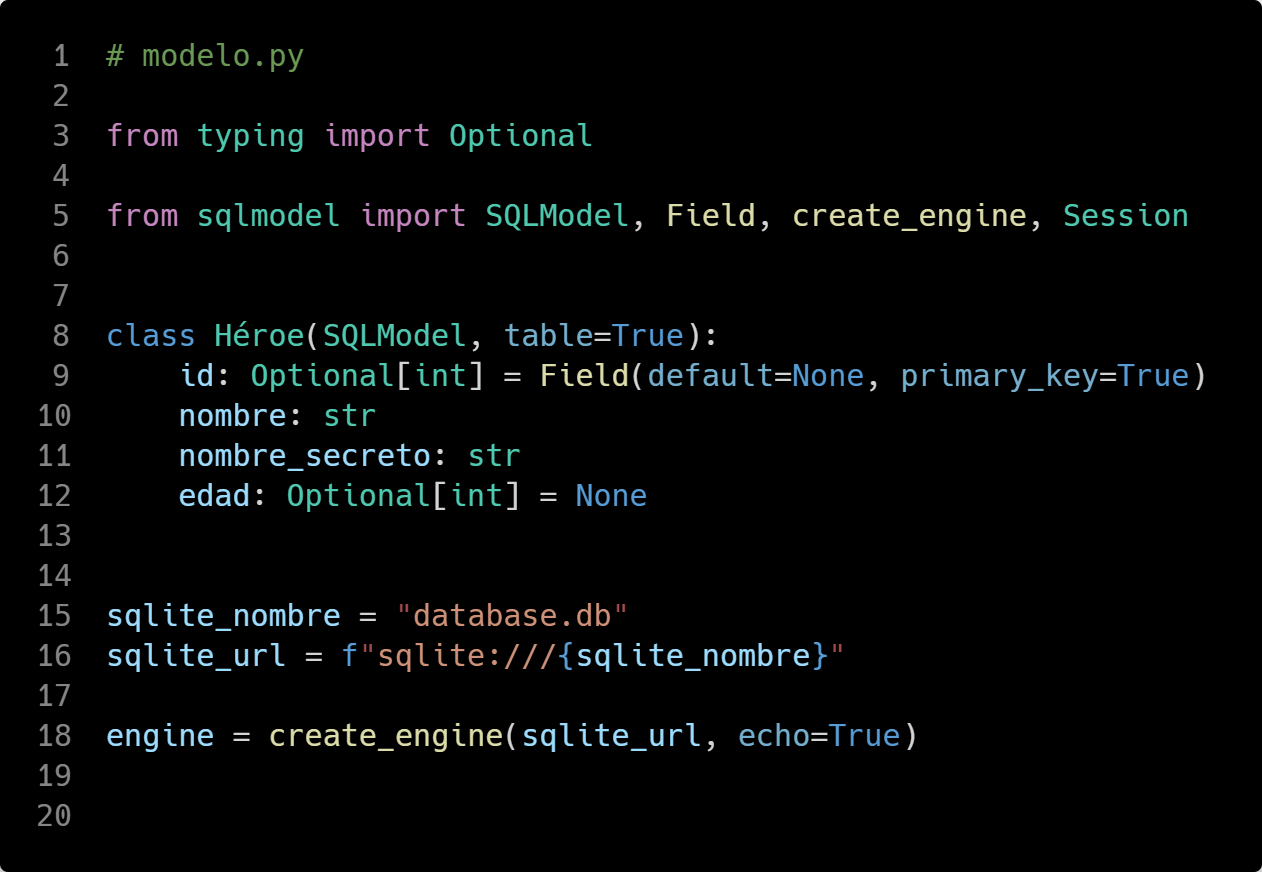
print("Héroe 1:", héroe\_1)

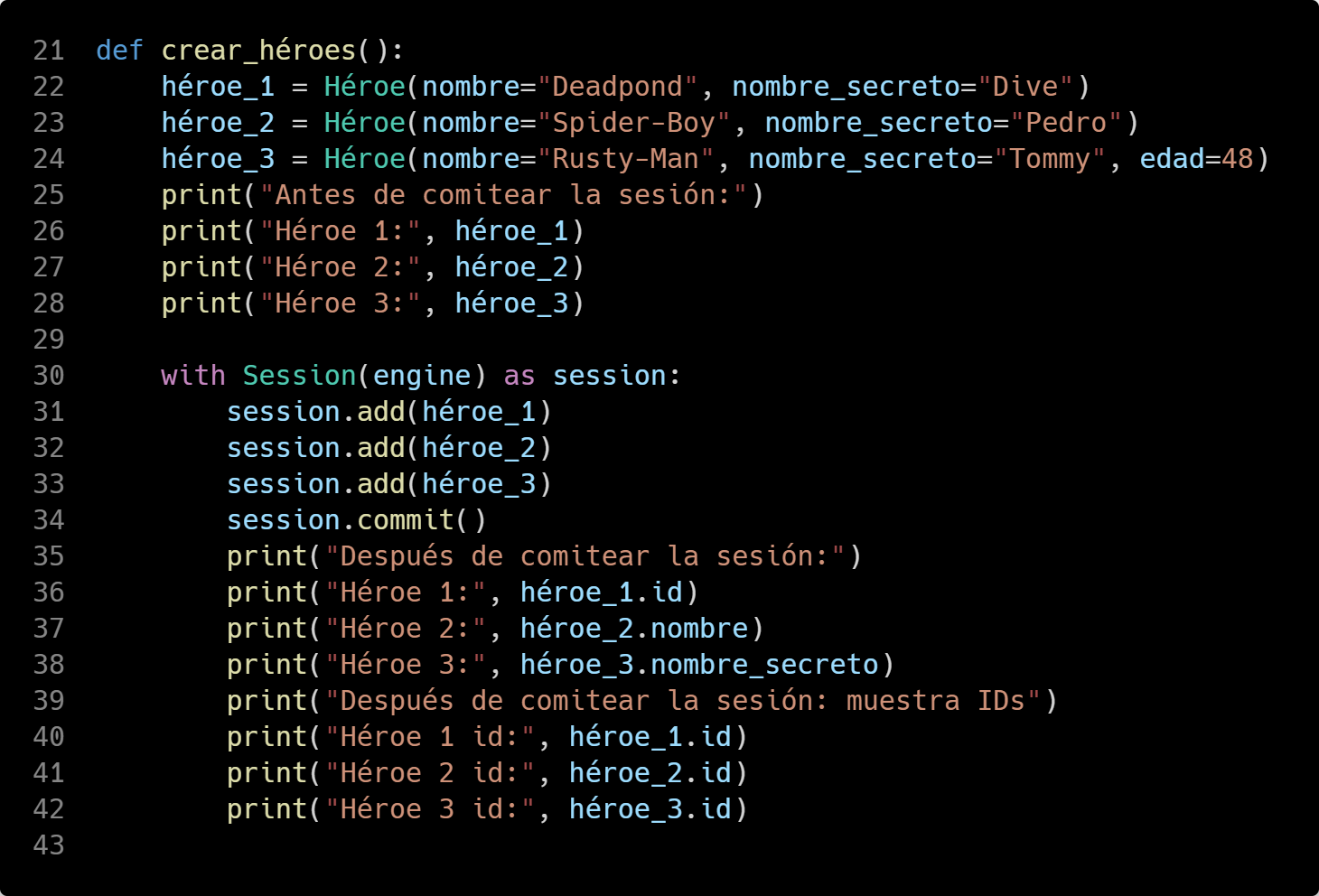
print("Héroe 2:", héroe\_2)

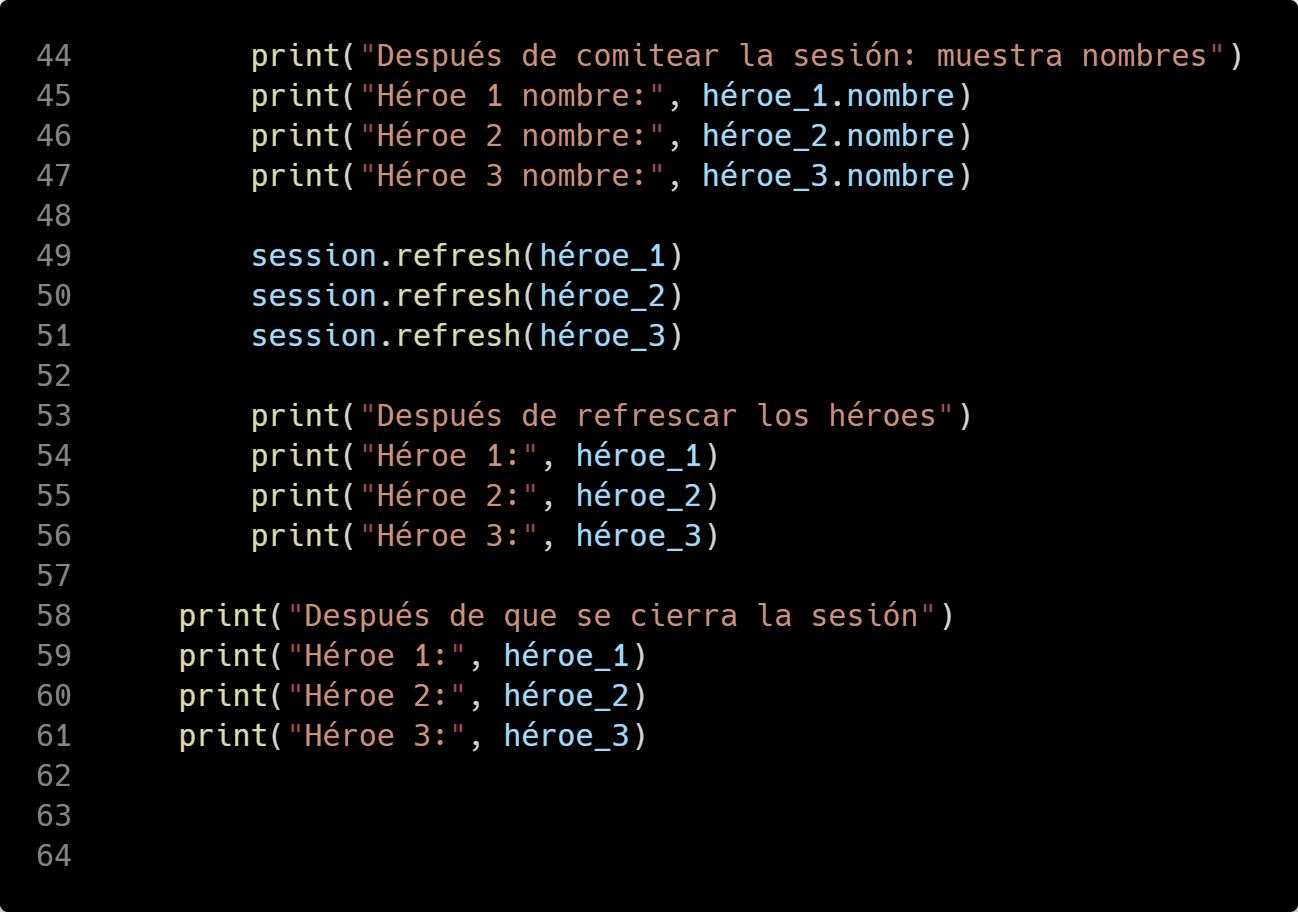
print("Héroe 3:", héroe\_3)

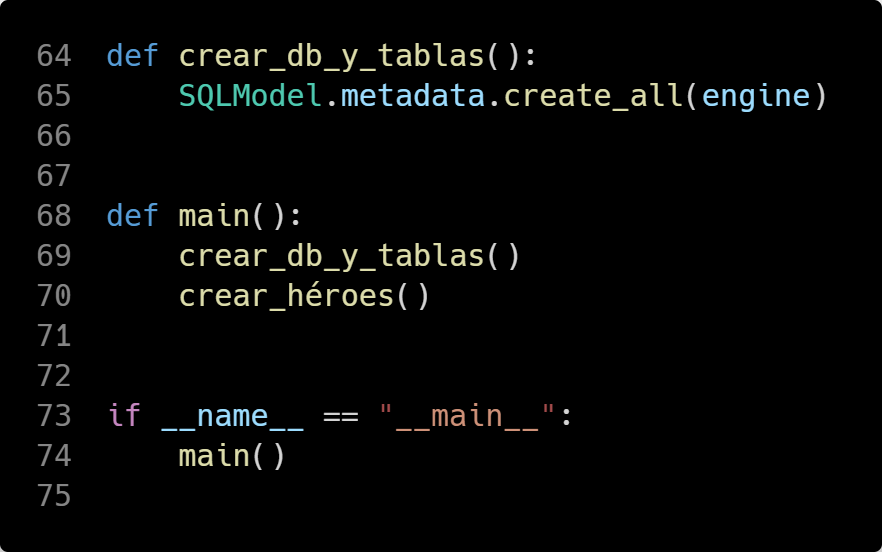
**¡MANOS A LA OBRA!** 

Ejecuta el siguiente código y observa todos los comportamientos descritos en la teoría.









## Revisemos lo aprendido hasta aquí

* Conocer las características de SQLModel
* Instalar SQLModel
* Crear tablas
* Comprender el modo de la creación de la clave primaria
* Comprender el funcionamiento esencial de SQLModel respecto al motor
* La importancia del orden de las importaciones
* Crear índices
* Crear la restricción unique para una columna

# Consultas (SELECT)

Hemos creado un modelo de clase SQLModel y hemos creado algunos registros en la base de datos. Para leer los datos necesitamos el modelo, el motor, y una nueva sesión para consultarlos.

Primero tenemos que importar select, luego crear una consulta con select, ejecutar la consulta con session.exec, guardando su devolución en un objeto que llamaremos "registros", que será iterable. Iteramos sobre el objeto e imprimimos sus elementos.

from sqlmodel import select

| Escribe la nueva función y llámala desde main(). No olvides importar select. |
| --- |

def consultar\_héroes():  
 with Session(engine) as session:  
 consulta = **select(**Héroe**)**

registros = **session.exec(**consulta**)**

for héroe in registros:

print(héroe)

Hasta ahora hemos estado usando los resultados para iterar sobre ellos. Pero, por diferentes razones, es posible que desees tener la lista completa de objetos Héroe de inmediato, en lugar de un iterable. Por ejemplo, si deseas devolverlos en una API web.

El objeto al que se le asigna session.exec(consulta), en este ejemplo llamado "registros", contiene el método all() que devuelve una lista con todos los objetos:

def consultar\_héroes():

with Session(engine) as session:

consulta = **select(**Héroe**)**

registros = **session.exec(**consulta**)**

héroes = registros.**all()**

print(héroes)

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

| Si el resultado es confuso, no olvides que puedes hacer:  from pprint import pprint  pprint(héroes) |
| --- |

Se han ido creando varias variables para poder explicar mejor qué va haciendo cada cosa. Pero sabiendo qué es cada objeto y qué hace cada uno, podemos simplificarlo un poco y ponerlo en una forma más compacta:

def consultar\_héroes():

with Session(engine) as session:

héroes = **session.exec(select**(Héroe)**).all()**

print(héroes)

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

## Filtrado - where

De la misma manera que agregamos **WHERE** a una instrucción SQL para filtrar filas, podemos agregar un **.where()** a una instrucción **select()** de SQLModel para filtrar filas en la consulta de datos.

def consultar\_héroes():  
 with Session(engine) as session:  
 sentencia = select(Héroe)**.where**(Héroe.nombre == "Deadpond")  
 registros = session.exec(sentencia)  
 for héroe in registros:  
 print(héroe)

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

### Múltiples where

Crear más instancias de Héroe y agrégalos a la sesión: session.add(...) Todo esto servirá para ejemplificar lo que viene a continuación.

héroe\_4 = Héroe(nombre="Tarantula",   
 nombre\_secreto="Natalia Roman-on", edad=32)

héroe\_5 = Héroe(nombre="Black Lion",   
 nombre\_secreto="Trevor Challa", edad=35)

héroe\_6 = Héroe(nombre="Dr. Weird",   
 nombre\_secreto="Steve Weird", edad=36)

héroe\_7 = Héroe(nombre="Captain North America",   
 nombre\_secreto="Esteban Rogelios", edad=93)

El objeto devuelto por select es un tipo especial de objeto con algunos métodos. Uno de esos métodos es where() usado para agregar una condición a ese objeto seleccionado. Select tiene más métodos aún, y la mayoría de estos métodos devuelven el mismo objeto después de modificarlo: gracias a esto, podemos llamar un método tras otro:

sentencia = select(Héroe)**.where**(Héroe.nombre != "Deadpond"  
 **).where**(Héroe.edad == 48)

| Puedes utilizar los operadores == != > < >= <= propios de Python, lo que es una ventaja frente a otros ORMs. |
| --- |

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

Como alternativa, también podemos pasar varias expresiones en una sola:

sentencia = select(Héroe)**.where(**Héroe.edad >= 35, Héroe.edad < 40**)**

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

Es posible que tu editor te marque un error cuando estés haciendo comparaciones de números. Esto se debe a que estamos usando anotaciones de Python para los campos, de hecho, la **edad** se anotó como **Optional[int] = None.**

Al usar estas anotaciones de tipo, obtenemos el beneficio de la simplicidad adicional y las comprobaciones de errores en sobre la marcha al crear o usar instancias.

Pero el editor no sabe que es un atributo de clase especial, por lo que intenta ayudarnos a prevenir un error (que en este caso es una falsa alarma). Sin embargo, podemos arreglarlo fácilmente:

Podemos decirle al editor que este atributo de clase es en realidad una columna especial de SQLModel (en lugar de un atributo de instancia con un valor normal).

Para hacer eso, importamos col (abreviatura de columna):

**from sqlmodel import col**

sentencia = select(Héroe).where(**col(**Héroe.edad**)** >= 35,  
 **col(**Héroe.edad**)** < 40)

| Modifica tu código según el código de arriba y observa cómo se comporta VSCode |
| --- |

**col** es útil para dar autocompletado a otros métodos como **contains**, que sirva para filtrar cadenas. El siguiente ejemplo filtra todos los héroes cuyo nombre contiene la letra "u":

sentencia = select(Héroe).where(**col(**Héroe.nombre**)**.**contains(**"u"**)**)

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

**col** tiene el método **is\_**, que sirve para filtrar objetos que en la base de datos son NULL

#### sentencia = select(Héroe).where(**col(**Héroe.edad**).is\_(**None**)**)

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

**col** tiene otros métodos especiales, como startswith, endswith, que actúan de manera similar a contains. Tienes que probarlos...

**col** tiene un método interesante como .in\_ , que recibe una colección (tupla, lista, conjunto, diccionario) y si el valor se encuentra en la colección, el registro es filtrado:

misiones = {"Batman": 123, "Tarantula": 323}  
sentencia = select(Héroe).where(**col(**Héroe.nombre)**.in\_(**misiones**)**)

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

Hay algunos métodos que col() no tiene, como es el de la longitud de cadena. Para poder medir la longitud de cadena y luego compararlo, debemos usar directamente SQLAlchemy:

**from** **sqlalchemy.sql.expression** **import** **func**  
sentencia = **select**(Héroe)**.filter(func.length(**Héroe.nombre\_secreto**)** < 10**)**

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

### 

### 

### 

### Múltiples where usando or\_

Podemos combinar expresiones usando or\_: cualquier expresión, si es verdadera en una fila (y no necesariamente en todas las expresiones), esta fila será incluida en los resultados.

from sqlmodel import or\_

sentencia = select(Héroe).where(**or\_(**col(Héroe.edad) >= 35,   
 col(Héroe.edad) > 90))

| Modifica tu código según el código de arriba y compara esta salida con la salida "sin" or\_ |
| --- |

### Lectura de una fila

En los ejemplos, al ejecutar un **select** hemos visto cómo devuelve un objeto iterable. O puedes llamar resultado.all() para obtener una lista de todas las filas de inmediato, en lugar de un iterable.

#### héroes = **session.exec(**select(Héroe)**).all()**

Pero si queremos únicamente la primera fila, podemos llamar a **.first()** en el objeto de resultados. Si no encuentra ninguna fila, devolverá **None**:

#### héroes = **session.exec(**select(Héroe)**).first()**

| Modifica tu código según el código de arriba, y ejecuta el programa |
| --- |

Puede existir el caso en el que queramos asegurarnos de que haya exactamente **una fila** que coincida con la consulta. Y si hubiera más de una, significaría que hay un error en el sistema de datos, y debería terminar con un **error**. Para tal caso, debemos usar **.one()**

resultado = session.exec(**select(**Héroe**)**.

where(Héroe.nombre == "Deadpond"))**.one()**

| Modifica tu código según el código de arriba. Si vuelves a crear 2 veces los mismos héroes, verás un error, ya que debería haber un solo héroe con el nombre "Deadpond". |
| --- |

### Select id

Si quieres seleccionar una sola fila por su columna id como la clave principal, podrías hacer lo siguiente:

#### consulta = select(Héroe).where(Héroe.id == 1)

#### resultado = session.exec(consulta).first()

Pero tienes un atajo con session **.get()**:

#### héroe = session.**get(**Hero, 1**)**

### Límites

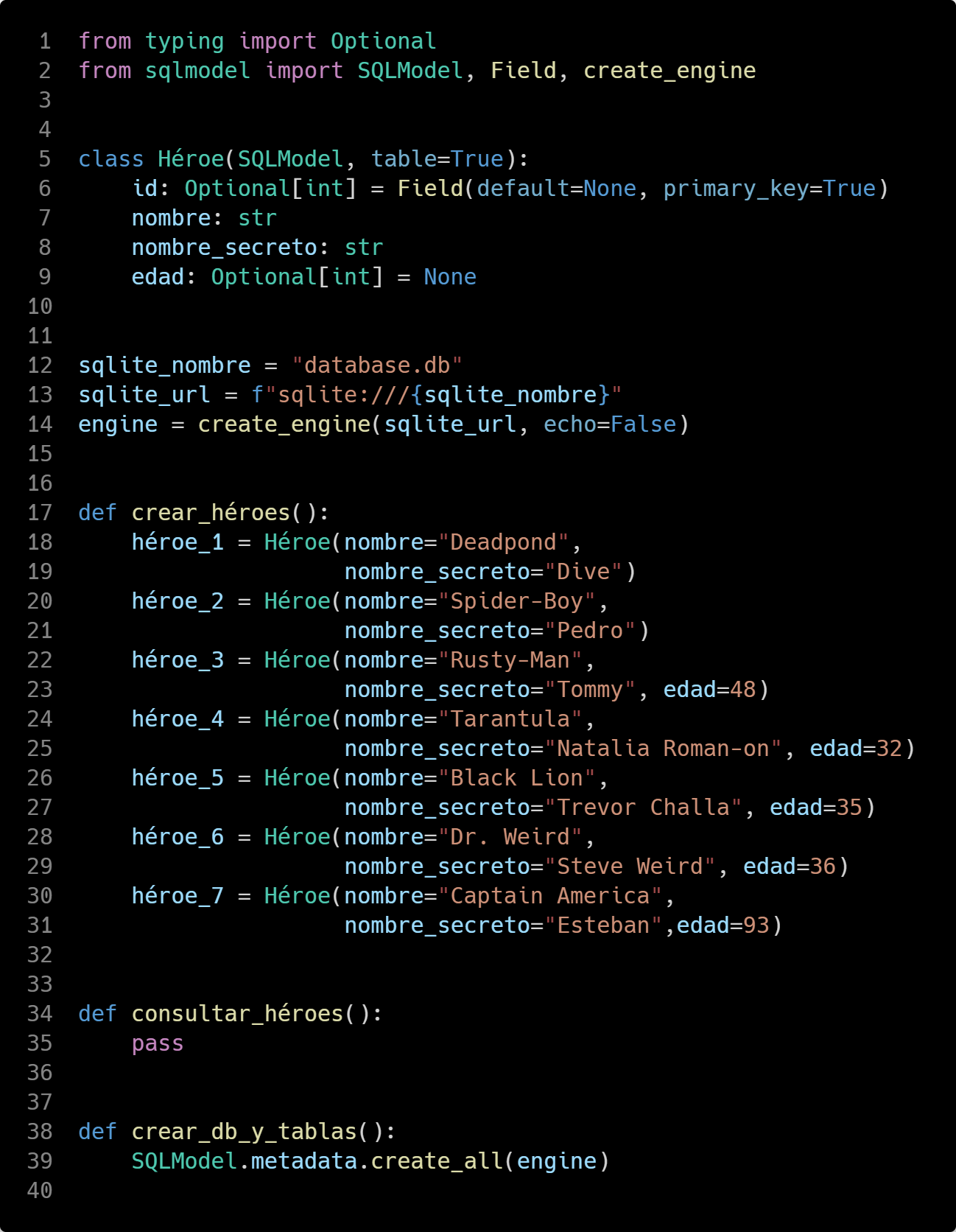
Para limitar los resultados tenemos el método select()**.limit()**:

#### consulta = **select(**Héroe**)**.where(col(Héroe.edad) > 40)**.limit(**2**)**

| Prueba limit() para una consulta de todos los héroes |
| --- |

**¡MANOS A LA OBRA!** 

A partir del código de abajo



1. Muestra los registros que contengan en su nombre el guion "-" y, a su vez, edad sea null.
2. Muestra los registros cuyo nombre sea mayor a 9 y edad menor a 70
3. Muestra el registro cuyo id sea 1.
4. Muestra los registros que no tengan edad o los que tengan edad mayor a 65.
5. Utiliza .get() como atajo para el ejercicio 3.

|  | **Ver Video “**[**Bases de datos parte 2**](https://www.youtube.com/watch?v=kGnVjcZ6RHA&list=PLgwlfcqa5h3zf1_JBC9RH59lf_g1yBGxK&index=2)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Hacer consultas con select
* Filtrar las consultas con where
* Realizar múltiples where
* Utilizar el operador or\_
* Lectura de una sola fila
* Usar el select para consultar un id
* Limitar la cantidad de registros en una consulta

# Modificación (UPDATE)

Para actualizar una fila debes:

1. Leer la base de datos y guardar los datos en el objeto
2. Establecer el nuevo valor del atributo del objeto
3. Agregar el objeto a la sesión
4. Comitear la sesión
5. Puedes refrescar el objeto y devolverlo al usuario

#### def modificar\_héroes(): with Session(engine) as session: consulta = select(Héroe).where(Héroe.nombre == "Spider-Boy") resultado = session.exec(consulta) **héroe = resultado.one()** print("Héroe:", héroe) **héroe.edad = 20** session.**add**(héroe) session.**commit()** session.refresh(héroe) print("Héroe modificado:", héroe)

**¡MANOS A LA OBRA!** 

Agrega el código de arriba a tu proyecto, y en la misma sesión cambiar 2 nombres y 2 edades de diferentes héroes.

# Eliminación (DELETE)

Si deseas eliminar un solo valor en una columna mientras quieres mantener la fila, debes actualizar la fila en NULL (None en Python).

Pero, **para eliminar una fila**, debes:

1. Leer la base de datos y guardar los datos en el objeto
2. Eliminar el objeto de la sesión
3. Comitear la sesión

#### def eliminar\_héroes(): with Session(engine) as session: consulta = select(Héroe).where(Héroe.nombre == "Tarantula") resultado = session.exec(consulta) **héroe = resultado.one()** print("Héroe:", héroe) **session.delete(héroe)** session.commit() print("Héroe eliminado:", héroe)

**¡MANOS A LA OBRA!** 

Crea una función que elimine todos los héroes cuya edad sea NULL usando un iterador. Muestra al usuario cuáles son los id y nombres que se eliminarán. Una vez que el usuario confirme la eliminación, luego, verifica si realmente se eliminaron en la tabla, usando la extensión SQLite, de Visual Studio Code.

## Revisemos lo aprendido hasta aquí

* Modificar filas con otro valor o establecerlas en null
* Eliminar filas con <Session>.delete()

## 

# Relaciones entre tablas

La principal característica de las bases de datos SQL es poder manejar datos relacionados, es decir, conectar o unir diferentes tablas.

El primer paso es crear más de una tabla y conectarlas, de modo que cada fila de una tabla pueda hacer referencia a otra fila en la otra tabla. Entonces, agreguemos la tabla "equipo" y agreguemos un campo más en la tabla "héroe", que hará referencia a la tabla "equipo", de modo tal que un héroe puede pertenecer a un equipo:

#### **class Equipo(SQLModel, table=True): id: Optional[int] = Field(default=None, primary\_key=True) nombre: str sede: str**

#### class Héroe(SQLModel, table=True): id: Optional[int] = Field(default=None, primary\_key=True) nombre: str = Field(index=True) nombre\_secreto: str edad: Optional[int] = None **equipo\_id: Optional[int] = Field(default=None, foreign\_key="equipo.id")**

Aquí estamos creando datos conectados en una relación en la que un equipo podría tener muchos héroes. Por lo tanto, comúnmente se denomina relación de "uno a muchos" o de "muchos a uno".

Observa que "equipo.id" hace referencia al nombre de la tabla de la base de datos, por lo que "equipo", no es el nombre de la clase modelo "Equipo", sino de la tabla.

| Agrega el código de arriba y vuelve a crear las tablas con la función que ya hemos definido. Observa la nueva estructura en el explorador de SQLite. |
| --- |

## Creación de tablas relacionadas

Primero, creamos dos equipos:

#### def crear\_héroes(): with Session(engine) as session: preventores = Equipo(nombre="Preventores", sede="La Torre") fuerza\_z = Equipo(nombre="Fuerza Z", sede="El Bar") session.add(preventores) session.add(fuerza\_z) session.commit()

Ahora, como la tabla "héroes" tiene un nuevo campo "equipo\_id", podemos configurarlo usando el campo de id de los objetos de equipo que acabamos de crear:

#### def crear\_héroes(): with Session(engine) as session: preventores = Equipo(nombre="Preventores", sede="La Torre") fuerza\_z = Equipo(nombre="Fuerza Z", sede="El Bar") session.add(preventores) session.add(fuerza\_z) session.commit() héroe\_deadpond = Héroe(nombre="Deadpond", nombre\_secreto="Dive", **equipo\_id=fuerza\_z.id)** héroe\_rusty\_man = Héroe(nombre="Rusty-Man", nombre\_secreto="Tommy", edad=48, **equipo\_id=preventores.id)** héroe\_spider\_boy = Héroe(nombre="Spider-Boy", nombre\_secreto="Pedro") session.add(héroe\_deadpond) session.add(héroe\_rusty\_man) session.add(héroe\_spider\_boy) session.commit()

| Borra el archivo de la base de datos, agrega el código de arriba y vuelve a ejecutar el programa. Observa los datos en el explorador de SQLite. |
| --- |

Puedes refrescar los objetos y mostrarlos al usuario:

#### session.refresh(héroe\_deadpond) session.refresh(héroe\_rusty\_man) session.refresh(héroe\_spider\_boy) print("Héroe creado:", héroe\_deadpond) print("Héroe creado:", héroe\_rusty\_man) print("Héroe creado:", héroe\_spider\_boy)

| Agrega el código después del último commit() y prueba... |
| --- |

## Consulta de relaciones

#### def consultar\_héroes(): with Session(engine) as session: consulta = **select(Héroe, Equipo). where(Héroe.equipo\_id == Equipo.id)** resultados = session.exec(consulta) for héroe, equipo in resultados: print("Héroe:", héroe, "Equipo:", equipo)

| Reescribe la función consultar\_héroes con el código de arriba y prueba la función |
| --- |

Puedes simplificarlo con **join**:

#### def consultar\_héroes(): with Session(engine) as session: consulta = **select(Héroe, Equipo).join(Equipo)** resultados = session.exec(consulta) for héroe, equipo in resultados: print("Héroe:", héroe, "Equipo:", equipo)

| Reescribe la función consultar\_héroes con el código de arriba y prueba la función |
| --- |

Puedes incluir los registros NULL con el parámetro isouter de join():

#### consulta = select(Héroe, Equipo).**join**(Equipo, **isouter=True**)

| Prueba isouter (en SQL es el equivalente a LEFT OUTER JOIN) para ver qué registros devuelve |
| --- |

## Modificación de relaciones

#### print("ALGUIEN SE UNIÓ A UN EQUIPO:") **héroe\_spider\_boy.equipo\_id = preventores.id** session.add(héroe\_spider\_boy) session.commit() session.refresh(héroe\_spider\_boy) print("Héroe modificado:", héroe\_spider\_boy)

| Agrega el código al final de la función crear\_héroes, y prueba... |
| --- |

## Eliminación de relaciones

Realmente no tenemos que borrar nada para romper una relación. Simplemente, asignamos **None** a la clave foránea:

#### print("VOLVIÓ A QUEDAR SIN EQUIPO:") **héroe\_spider\_boy.equipo\_id = None** session.add(héroe\_spider\_boy) session.commit() session.refresh(héroe\_spider\_boy) print("Héroe modificado:", héroe\_spider\_boy)

| Agrega el código al final de la función crear\_héroes, y prueba... |
| --- |

## Revisemos lo aprendido hasta aquí

* Crear tablas relacionadas
* Consultar relaciones
* Modificar relaciones
* Eliminar relaciones

# Atributos de relación

Anteriormente, habíamos creado algunos objetos de instancia de **Equipo**. También usamos los id de estos equipos al crear las instancias de **Héroe**, en el campo **equipo\_id**. Pero las instancias del modelo no tienen un **id** generado por la base de datos **hasta que** las agregamos y las comiteamos en la sesión. Antes de eso, son **None**. Por lo tanto, primero tuvimos que agregarlos (**add**) y luego confirmar (**commit**). Entonces recién pudimos crear las instancias de Héroe para poder usar sus **id**. Agregamos los nuevos héroes a la sesión y luego los comiteamos.

**¿Estamos comiteando dos veces?**

Debemos recordar, agregar algunas cosas primero, confirmar, y hacer todo eso en el orden correcto, de lo contrario, podríamos terminar usando, equivocadamente, un equipo.id = None porque nos olvidamos comitearlo. Esto es lo primero en donde los atributos de relación pueden ayudar

### ¿Qué son los atributos de relación?

Los atributos no son lo mismo que campos, **no representan una columna** directamente en la base de datos, y su valor no es un valor singular como un número entero. Su valor **es el objeto** que está relacionado.

Con un atributo de relación podrás llamar a una instancia héroe de la siguiente manera: **héroe.equipo**, y obtendrás el objeto de la instancia "equipo" completo al que pertenece este héroe.

Esto sirve, por ejemplo, para verificar si un héroe pertenece a algún equipo (si equipo no es None) y luego imprimir el nombre del equipo:

#### if héroe.equipo: print(héroe.equipo.nombre)

## Definición de atributos de relación

#### from sqlmodel import Relationship

#### class Equipo(SQLModel, table=True): id: Optional[int] = Field(default=None, primary\_key=True) nombre: str sede: str **héroes: list["Héroe"] = Relationship(back\_populates="equipo")**

#### class Héroe(SQLModel, table=True): id: Optional[int] = Field(default=None, primary\_key=True) nombre: str = Field(index=True) nombre\_secreto: str edad: Optional[int] = None equipo\_id: Optional[int] = Field(default=None, foreign\_key="equipo.id") **equipo: Optional["Equipo"] = Relationship(back\_populates="héroes")**

| Modifica tu código como el que está arriba y vuelve a crear la tabla |
| --- |

**back\_populates:** Le dice a SQLModel que si algo cambia en este modelo, debería cambiar ese atributo en el otro modelo, y funcionará incluso antes de comitearse con la sesión.

## 

## Creación de atributos de relación

Ahora podemos crear las **instancias de Equipo** y pasarlas directamente al nuevo argumento **equipo** cuando creamos **las instancias de Héroe**. Es decir, ahora hacemos: **equipo=preventores**, en lugar de hacer: **equipo\_id=preventores.id**

#### def crear\_héroes(): with Session(engine) as session: preventores = Equipo(nombre="Preventores", sede="La Torre") fuerza\_z = Equipo(nombre="Fuerza Z", sede="El Bar") **~~session.add(preventores)~~ ~~session.add(fuerza\_z)~~ ~~session.commit()~~** héroe\_deadpond = Héroe(nombre="Deadpond", nombre\_secreto="Dive", **equipo=fuerza\_z)** héroe\_rusty\_man = Héroe(nombre="Rusty-Man", nombre\_secreto="Tommy", edad=48, equipo=preventores) héroe\_spider\_boy = Héroe(nombre="Spider-Boy", nombre\_secreto="Pedro") session.add(héroe\_deadpond) session.add(héroe\_rusty\_man) session.add(héroe\_spider\_boy) session.commit()

| Modifica tu código como el que está arriba y prueba… |
| --- |

Gracias a SQLAlchemy, estos equipos ni siquiera tienen que tener una identificación todavía, porque estamos asignando todo el objeto a cada héroe. Esos equipos se crearán automáticamente en la base de datos, y, además, se generará la identificación automática, y se establecerá en la columna equipo\_id para cada una de las filas de héroes correspondientes.

De hecho, ahora ni siquiera tenemos que poner los equipos explícitamente en la sesión con **session.add(preventores)**, porque estas instancias de equipo ya están asociadas con los héroes que agregamos a la sesión.

Luego, como puedes ver, solo tenemos que hacer **un solo commit()**.

## Asignación de una relación

#### héroe\_spider\_boy.equipo = preventores

#### session.add(héroe\_spider\_boy)

#### session.commit()

#### session.refresh(héroe\_spider\_boy)

#### print("Héroe modificado:", héroe\_spider\_boy)

| Modifica tu código como el que está arriba y prueba… |
| --- |

También podríamos crear varias instancias de Héroe y luego pasarlas a héroes= que es un argumento que recibe una lista cuando creamos una instancia de Equipo.

#### héroe\_león = Héroe(nombre="León Negro",

#### nombre\_secreto="Leny", edad=35)

#### héroe\_princesa = Héroe(nombre="Princesa Segura",

#### nombre\_secreto="Priny")

#### equipo\_agua = **Equipo**(nombre="Aqua", sede="AquaCity",

#### **héroes=[héroe\_león, héroe\_princesa])**

#### session.add(equipo\_agua)

#### session.commit()

#### session.refresh(equipo\_agua)

#### print("Equipo Aqua:", equipo\_agua)

| Modifica tu código como el que está arriba y prueba… |
| --- |

Observa que, al igual que antes, solo tenemos que agregar la instancia del Equipo a la sesión, y debido a que los héroes están relacionados con ella, también se guardarán automáticamente cuando comitemos.

### Incluir objetos de relación en el lado muchos

Esta es una relación de muchos a uno, porque puede haber muchos héroes que pertenecen a un equipo. También podemos relacionar datos con estos atributos de relación **en el lado de "muchos"**.

Como el atributo **equipo.héroes** se comporta como una lista, simplemente podemos agregarlo. Vamos a crear algunos héroes más y agregarlos al atributo de lista **preventores.héroes**:

#### héroe\_tarantula = Héroe(nombre="Tarantula", nombre\_secreto="Natalia Roman-on", edad=32) héroe\_dr\_weird = Héroe(nombre="Dr. Weird", nombre\_secreto="Steve Weird", edad=36) héroe\_capitán = Héroe(nombre="Captain America", nombre\_secreto="Esteban Rogelios", edad=93) **preventores.héroes.append(héroe\_tarantula) preventores.héroes.append(héroe\_dr\_weird) preventores.héroes.append(héroe\_capitán) session.add(preventores)** session.commit() session.refresh(héroe\_tarantula) session.refresh(héroe\_dr\_weird) session.refresh(héroe\_capitán) print("Núevo héroe de preventores:", héroe\_tarantula) print("Núevo héroe de preventores:", héroe\_dr\_weird) print("Núevo héroe de preventores:", héroe\_capitán)

| Modifica tu código como el que está arriba y prueba… |
| --- |

## Consulta de relaciones

La forma de consultar dos tablas relacionadas que venimos haciendo es la siguiente:

#### with Session(engine) as session: consulta = select(Héroe).where(Héroe.nombre == "Spider-Boy") resultado = session.exec(consulta) héroe\_spider\_boy = resultado.one() **consulta = select(Equipo).where( Equipo.id == héroe\_spider\_boy.equipo\_id)** resultado = session.exec(consulta) **equipo = resultado.first()** print("Equipo de Spider-Boy:", equipo)

Con los atributos de relación, se hace más simple:

#### with Session(engine) as session: consulta = select(Héroe).where(Héroe.nombre == "Spider-Boy") resultado = session.exec(consulta) héroe\_spider\_boy = resultado.one() print("Equipo de Spider-Boy:", **héroe\_spider\_boy.equipo**)

| Modifica tu código como el que está arriba y prueba… |
| --- |

## Eliminación de relaciones

#### def modificar\_héroes(): with Session(engine) as session: consulta = select(Héroe).where(Héroe.nombre == "Spider-Boy") resultado = session.exec(consulta) **héroe\_spider\_boy.equipo = None** print("Equipo de Spider-Boy:", héroe\_spider\_boy.equipo)

| Agrega lo de arriba y prueba (no olvides llamar la función modifica\_héroes desde main) |
| --- |

## Atajos en las consultas

La forma que hemos estado usando es la siguiente:

#### with Session(engine) as session: consulta = select(Héroe).where(Héroe.nombre == "Spider-Boy") resultado = session.exec(consulta) héroe\_spider\_boy = resultado.one()

Podemos usar una forma más corta:

#### with Session(engine) as session: héroe\_spider\_boy = **session.exec( select(**Héroe**).where(**Héroe.nombre == "Spider-Boy"**) ).one()**

## Revisemos lo aprendido hasta aquí

* Crear atributos de relación
* Modificar atributos de relación
* Eliminar atributos de relación

# Relación muchos a muchos

Supongamos que nuestros héroes pueden pertenecer a varios equipos a la vez. Necesitamos una tercera tabla: la vamos a llamar "HéroeEquipoLink", que contendrá dos claves foráneas, una para Héroe y otra para Equipo:

#### class HéroeEquipoLink(SQLModel, table=True):

#### equipo\_id: Optional[int] = Field(

#### default=None, foreign\_key="equipo.id", primary\_key=True)

#### héroe\_id: Optional[int] = Field(

#### default=None, foreign\_key="héroe.id", primary\_key=True)

| Agrega el código **antes** que las clases Equipo y Héroe |
| --- |

Ahora debemos hacer unos pequeños cambios:

#### class Equipo(SQLModel, table=True):

#### id: Optional[int] = Field(default=None, primary\_key=True)

#### nombre: str

#### sede: str

#### héroes: list["Héroe"] = Relationship(

#### back\_populates="equipos", link\_model=HéroeEquipoLink)

#### class Héroe(SQLModel, table=True):

#### id: Optional[int] = Field(default=None, primary\_key=True)

#### nombre: str = Field(index=True)

#### nombre\_secreto: str

#### edad: Optional[int] = None

#### equipo\_id: Optional[int] = Field(default=None, foreign\_key="equipo.id")

#### equipos: Optional["Equipo"] = Relationship(

#### back\_populates="héroes", link\_model=HéroeEquipoLink)

#### 

| Agrega los cambios y vuelve a crear las tablas |
| --- |

### Creación de datos

#### def crear\_héroes(): with Session(engine) as session: preventores = Equipo(nombre="Preventores", sede="La Torre") fuerza\_z = Equipo(nombre="Fuerza Z", sede="El Bar") héroe\_deadpond = Héroe(nombre="Deadpond", nombre\_secreto="Dive", **equipos=[fuerza\_z]**) héroe\_rusty\_man = Héroe(nombre="Rusty-Man", nombre\_secreto="Tommy", edad=48, **equipos=[preventores, fuerza\_z]**) héroe\_spider\_boy = Héroe(nombre="Spider-Boy", nombre\_secreto="Pedro", **equipos=[preventores]**) session.add(héroe\_deadpond) session.add(héroe\_rusty\_man) session.add(héroe\_spider\_boy) session.commit()

#### def consultar\_héroes(): with Session(engine) as session: héroes = session.exec(select(Héroe)).all() for héroe in héroes: print(héroe.nombre, héroe.equipos)

### Modificación de datos

#### def modificar\_héroes(): with Session(engine) as session: héroe\_spider\_boy = session.exec( select(Héroe).where(Héroe.nombre == "Spider-Boy") ).one() fuerza\_z = session.exec( select(Equipo).where(Equipo.nombre == "Fuerza Z") ).one() fuerza\_z.héroes.append(héroe\_spider\_boy) session.add(fuerza\_z) session.commit() print("Equipos de Spider-Boy:", héroe\_spider\_boy.equipos) print("Héroes de Fuerza Z:", fuerza\_z.héroes)

| Debido a que estamos accediendo a un atributo justo después de la confirmación, los datos se "refrescan" de manera implícita. |
| --- |

## El modelo link con campos extras

En el ejemplo anterior nunca interactuamos directamente con el modelo, todo fue a través de la relación automática de muchos a muchos a través de HéroeEquipoLink.

Pero, ¿qué pasaría si necesitáramos tener datos adicionales para describir el vínculo entre los dos modelos?

Digamos que queremos tener una columna más para decir si un héroe está entrenando en ese equipo.

Actualiza el modelo **HéroeEquipoLink**, y agrega al final:

#### **está\_entrenando: bool** equipo: "**Equipo**" = **Relationship**(back\_populates="**héroe\_links**") héroe: "**Héroe**" = **Relationship**(back\_populates="**equipo\_links**")

Actualiza la relación del modelo **Equipo**:

#### **héroe\_links**: list["**HéroeEquipoLink**"]=Relationship(back\_populates="**equipo**")

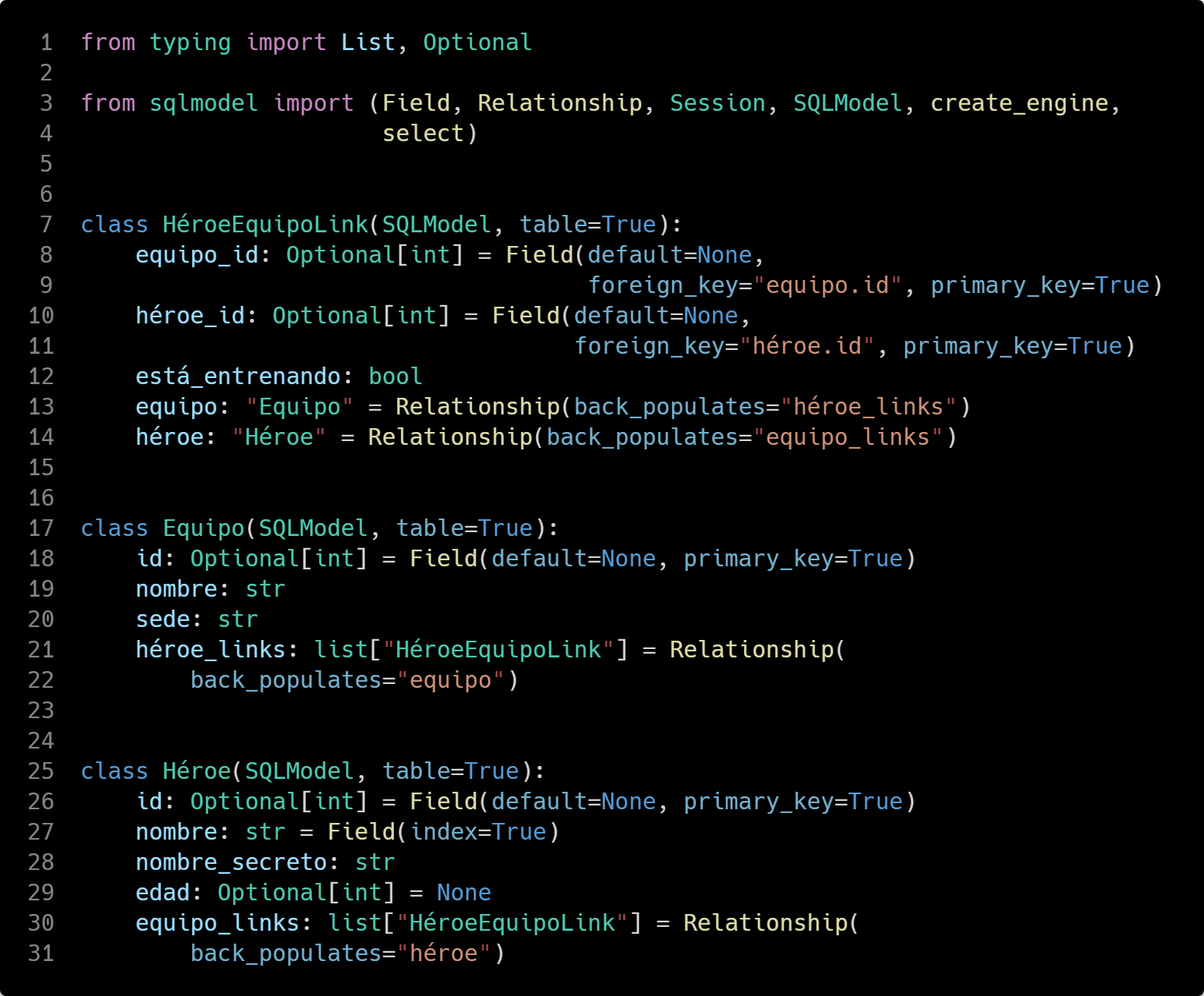
Actualiza la relación del modelo **Héroe**:

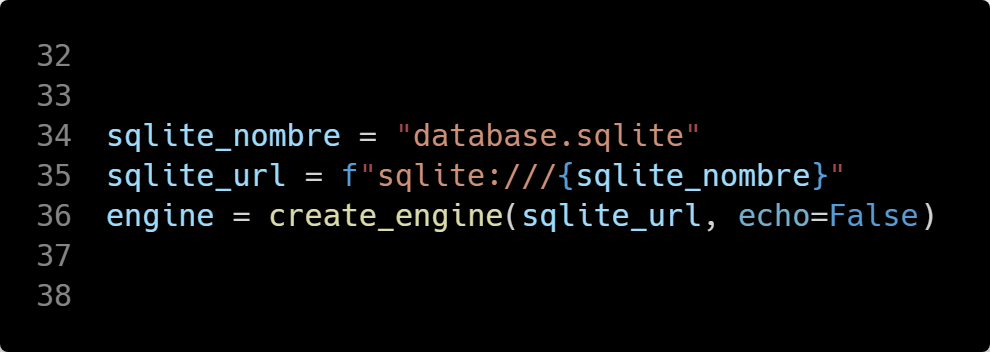
#### **equipo\_links**: list["**HéroeEquipoLink**"]=Relationship(back\_populates="**héroe**")

Ahora creamos los modelos Link manualmente, apuntando a sus instancias de héroe y equipo, y especificando los datos de link adicionales.

Aquí el código completo:

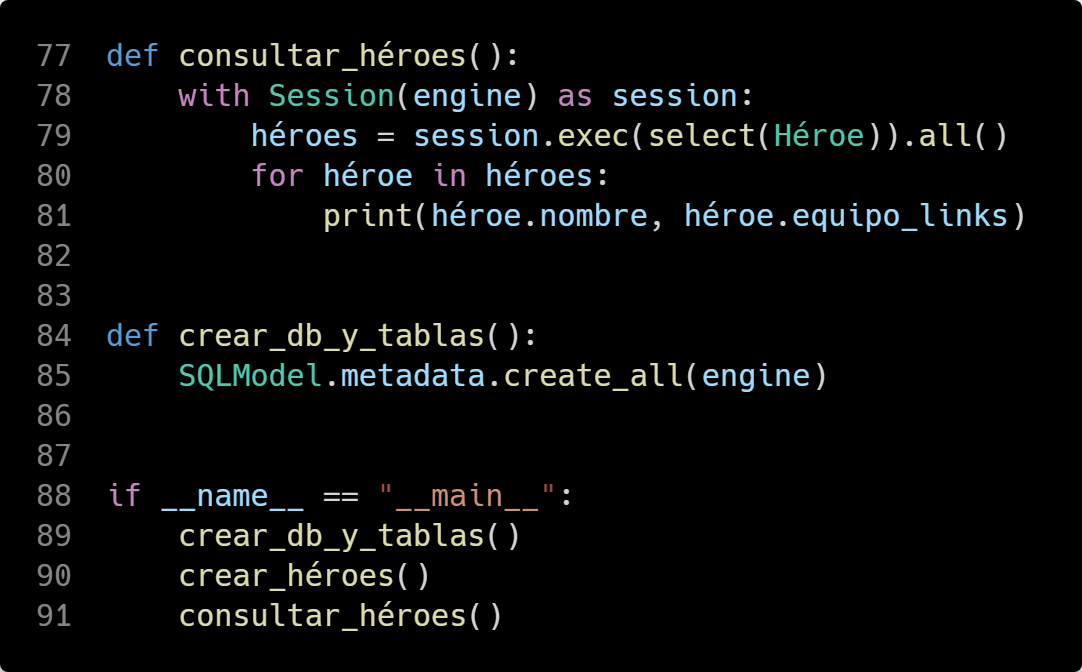
### Modelos





### Manejo de datos





## Revisemos lo aprendido hasta aquí

* Creación de datos en la relación muchos a muchos
* Modificar datos en la relación muchos a muchos
* Usar modelos de relación muchos a muchos con campos extras

# Decimales y más validaciones

En algunos casos, es posible que debas almacenar números decimales con garantías sobre la precisión. Es importante si está almacenando monedas, precios, cuentas ya que quisieras que no tuviera errores de redondeo.

**Abre el intérprete de Python y ejecuta:**

#### >>> 1.1 + 2.2

#### 3.3000000000000003

Esto se debe a la forma en que los números se almacenan en "unos y ceros" (tema del sistema binario y el cálculo de la coma flotante).

Debido a que las bases de datos almacenan datos de la misma manera que las computadoras (en binario), tendrían los mismos tipos de problemas. Debido a eso, también tienen un tipo decimal especial. En la mayoría de los casos, esto probablemente no sería un problema, por ejemplo, medir las vistas en un video o la barra de vida en un videojuego. Pero como puedes imaginar, esto es importante cuando se trata de dinero y finanzas.

#### from pydantic import condecimal

#### class Héroe(SQLModel, table=True): id: Optional[int] = Field(default=None, primary\_key=True) nombre: str = Field(index=True) nombre\_secreto: str edad: Optional[int] = None dinero: condecimal(max\_digits=5, decimal\_places=3) = Field(default=0)

Aquí estamos diciendo que el dinero puede tener como máximo 5 dígitos, esto incluye los números enteros (a la izquierda del punto decimal) y los decimales (a la derecha del punto decimal). También estamos diciendo que el número de decimales (a la derecha del punto decimal) es 3, por lo cual, podemos tener 3 dígitos decimales para estos números en el campo. Esto significa que tendremos 2 dígitos para la parte entera y 3 dígitos para la parte decimal.

Entonces, por ejemplo, estos son todos números válidos para el campo "dinero":

#### 12.345 12.3 12 1.2 0.123 0

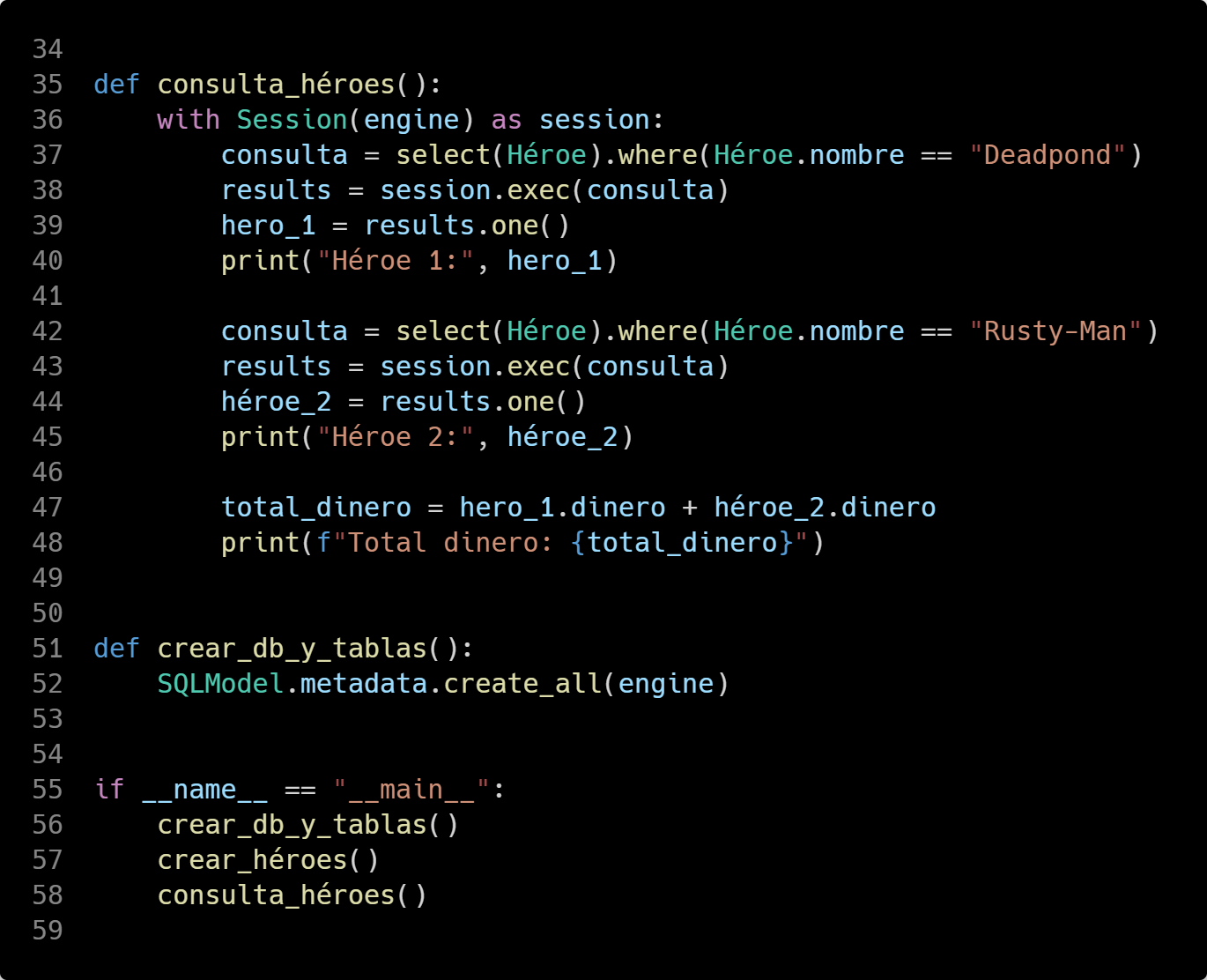
Pero los siguientes son números inválidos:

1.2345 tiene más de 3 decimales  
 123.234 tiene más de 5 dígitos en total (entero y parte decimal)  
 123 tiene más de 2 lugares en la parte entera

**¡MANOS A LA OBRA!** 

Prueba el siguiente código y prueba modificar los valores del campo dinero para cada instancia:





### Más validaciones

Pydantic tiene varios tipos de campos para validar, y muchas funcionalidades. Puedes ver la documentación original en:

<https://pydantic-docs.helpmanual.io/usage/types/>

# Estructura del código

Ten cuidado con las importaciones circulares. En nuestro ejemplo, la clase Héroe tiene una referencia a la clase Equipo internamente. Pero la clase Equipo también tiene una referencia a la clase Héroe. Por lo tanto, si esas dos clases estuvieran en archivos separados y trataras de importar las clases directamente en el archivo del otro, acabarás en una importación circular. Y Python no podrá manejarlo y arrojará un error.

Veamos diferentes estrategias para estructurar el código.

### La forma más simple

En este primer caso, todos los modelos vivirían en un solo archivo. La estructura de archivos del proyecto podría ser la siguiente:

.  
├── project  
 ├── \_\_init\_\_.py  
 ├── app.py  
 ├── database.py  
 └── models.py

Tenemos 3 módulos Python (o archivos) : app, database y models. También tenemos un \_\_init\_\_ vacío para hacer de este proyecto un "paquete de Python" (una colección de módulos de Python). De esta manera podemos usar importaciones relativas en el módulo app, como por ejemplo:

#### **\_\_init\_\_.py**

#### from .database import engine from .models import Héroe, Equipo

| Recuerda: podemos usar estas importaciones relativas porque en el módulo app, Python sabe que es parte de nuestro paquete, porque está en el mismo directorio que el archivo \_\_init\_\_. Y todos los archivos de Python en el mismo directorio también forman parte del mismo paquete. |
| --- |

Recuerda: podemos usar estas importaciones relativas porque en el módulo app, Python sabe que es parte de nuestro paquete, porque está en el mismo directorio que el archivo \_\_init\_\_. Y todos los archivos de Python en el mismo directorio también forman parte del mismo paquete

**models.py**

from typing import Optional  
from sqlmodel import Field, Relationship, SQLModel

class **Equipo**(SQLModel, table=True):

…

class **Héroe**(SQLModel, table=True)

...

#### **database.py**

#### from sqlmodel import SQLModel, create\_engine

#### sqlite\_file\_name = "database.db" sqlite\_url = f"sqlite:///{sqlite\_file\_name}" engine = create\_engine(sqlite\_url)

#### def crear\_db\_y\_tablas(): SQLModel.metadata.create\_all(engine)

#### **app.py**

#### from sqlmodel import Session from .database import engine, crear\_db\_y\_tablas **from .models import Héroe, Equipo**

#### def crear\_héroes(): ...

#### def main(): ...

#### if \_\_name\_\_ == "\_\_main\_\_": main()

### Separando los modelos por archivo

Si por alguna razón no quieres tener todos los modelos de la de base de datos juntos en un solo archivo, y los quieres tener separados, podríamos tener la siguiente estructura:

├── project  
 ├── \_\_init\_\_.py  
 ├── app.py  
 ├── database.py  
 ├── **model\_héroe.py** └── **model\_equipo.py**

El problema con las importaciones circulares es que Python no puede resolverlas en "tiempo de ejecución". Cuando se usan **anotaciones de tipo**, es muy común tener que declarar el tipo de algunas variables con clases importadas de otros archivos. Es posible que los archivos con esas clases también necesiten importar más elementos de los primeros archivos. Y esto termina requiriendo las mismas importaciones circulares que no son compatibles con Python en tiempo de ejecución.

Para resolverlo, hay un truco especial con una variable especial **TYPE\_CHECKING** del módulo **typing**. Tiene un valor de True para los editores y herramientas que analizan el código siguiendo las anotaciones de tipo; pero cuando Python se está ejecutando, su valor es False. Entonces, podemos usar un bloque if e importar dentro del bloque, pues solo será "importado" para el editor, pero no en tiempo de ejecución.

**\_\_init\_\_.py**

from .database import engine  
 from .model\_héroe import Héroe  
 from .model\_equipo import Equipo

**model\_héroe.py**

**from typing import** Optional, **TYPE\_CHECKING**  
 **if TYPE\_CHECKING:  
 from .model\_equipo import Equipo**

from sqlmodel import Field, Relationship, SQLModel

class **Héroe**(SQLModel, table=True):  
 ...

**model\_equipo.py**

**from typing import** Optional, **TYPE\_CHECKING**  
 **if TYPE\_CHECKING:  
 from .model\_héroe import Héroe**

from sqlmodel import Field, Relationship, SQLModel

class **Equipo**(SQLModel, table=True):  
 ...

**app.py**

from sqlmodel import Session  
 from .database import engine, crear\_db\_y\_tablas   
 **from .model\_héroe import Héroe  
 from .model\_equipo import Equipo**

def crear\_héroes():  
 ...

def main():  
 ...

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

## Ejecucción

Debido a que ahora, por ser un paquete de Python, este es un proyecto más grande, y no un solo archivo de Python, no podemos llamarlo simplemente pasando un solo nombre de archivo como lo hicimos antes: **python app.py**

Ahora tenemos que decirle a Python que queremos que ejecute un módulo que es parte de un paquete:

#### **python -m project.app**

El modificador **-m** es para decirle a Python que llame a un **módulo**. Y luego le pasamos **project.app**, que es el mismo formato que usamos en una importación: import project.app

Python ejecutará ese módulo dentro de ese paquete, y debido a que Python lo está ejecutando directamente, el mismo truco con el bloque principal que tenemos en app.py seguirá funcionando:

if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# 

# Funcionalidades avanzadas

Es necesario que sigas viendo otras características, como la integridad referencial cuando eliminas registros: si los eliminas en cascada o estableces un valor NULL al eliminar un campo que es la clave foránea del campo de otra tabla.

**Puedes ver la documentación de SQLAlchemy:**  <https://docs.sqlalchemy.org/>