SQLAlchemy Python SQL toolkit

START HERE

`create_engine("<url>")`
Connect to a database

`class User(Base)`
Map a table to a class

`Base.metadata.create_all(engine)`
Create the tables

`with Session(engine) as s:`
Open a unit of work

`s.add(obj)`
Queue an INSERT

`s.scalars(select(User))`
Query objects

`s.commit()`
Flush and save

`relationship()`
Link two classes

Engine URL + pool → `connect` → **Connection** Core, raw SQL → `wraps` → **Session** ORM unit of work → `tracks` → **Mapped objects** identity map

CONCEPTS

Core is SQL as Python expressions (Engine, Connection, Table). The ORM adds mapped classes and the Session on top.

Nothing is saved until commit(). Leaving the with-block without commit rolls back.

ENGINE

`create_engine("sqlite:///app.db")`
SQLite file engine

`create_engine("postgresql+psycopg://<user>:<pw>@<host>/<db>")`
PostgreSQL engine
Format is dialect+driver://user:password@host:port/database.

`create_engine("sqlite://", echo=True)`
In-memory DB, log all SQL

`from sqlalchemy import text`
Wrap a raw SQL string

`with engine.connect() as conn:`
Core connection block
Autobegin: call conn.commit() to save; nothing is committed on exit.

`conn.execute(text("SELECT 1"))`
Run raw SQL

ENGINE (cont.)

`conn.execute(text("... :id"), {"id": 5})`
Bind parameters by name
Never format values into the string; use bound parameters.

`with engine.begin() as conn:`
Connection that commits on exit

DECLARE MODELS

`class Base(DeclarativeBase): pass`
Declarative base class

`__tablename__ = "user"`
Table name for a model

`id: Mapped[int] = mapped_column(primary_key=True)`
Primary key column

`name: Mapped[str] = mapped_column(String(50))`
Typed string column

`email: Mapped[Optional[str]]`
Nullable column
Mapped[str] means NOT NULL; Optional makes it nullable.

`mapped_column(unique=True, index=True)`
Unique constraint / index

DECLARE MODELS (cont.)

`mapped_column(ForeignKey("user.id"))`
Foreign key column

`Base.metadata.create_all(engine)`
Create missing tables
Does not alter existing tables; use Alembic for that.

SESSION

`with Session(engine) as session:`
Open a session

`session.add(<obj>)`
Add one object

`session.commit()`
Write changes and end transaction

`session.rollback()`
Discard pending changes

`session.get(User, <pk>)`
Fetch by primary key

`sessionmaker(engine)`
Session factory for an app

QUERY

`session.scalars(select(User)).all()`
All rows as objects

`select(User).where(User.name == "ann")`
Filter rows

QUERY (cont.)

`session.scalars(stmt).first()`
First object or None

`session.execute(select(User.id, User.name))`
Rows of column tuples

`select(User).order_by(User.name.desc())`
Sort rows

`.where(or_(<a>, <b>))`
OR conditions

`select(Order.user_id, func.sum(Order.total)).group_by(Order.user_id)`
Aggregate per group

INSERT, UPDATE, DELETE

`user.name = "bob"`
Update via attribute, then commit

`update(User).where(<cond>).values(name="x")`
Bulk UPDATE

RELATIONSHIPS

`posts: Mapped[list["Post"]] = relationship(back_populates="author")`
One-to-many side

`author: Mapped["User"] = relationship(back_populates="posts")`
Many-to-one side

RELATIONSHIPS (cont.)

`relationship(cascade="all, delete-orphan")`
Delete children with parent

`select(User).options(selectinload(User.posts))`
Eager-load collection
Avoids the N+1 query pattern.

`select(Post).options(joinedload(Post.author))`
Eager-load by JOIN

JOINS

`select(User, Post).join(Post)`
Inner join on the foreign key

`select(User).join(User.posts)`
Join along a relationship

ALEMBIC MIGRATIONS

`alembic init <dir>`
Create migration environment

`target_metadata = Base.metadata`
Set in env.py for autogenerate

`alembic revision --autogenerate -m "<msg>"`
Diff models against DB
Review the generated file: renames show up as drop + add.

`alembic upgrade head`
Apply all migrations

ALEMBIC MIGRATIONS (cont.)

`alembic downgrade -1`
Undo the last migration

RECIPES

Minimal ORM app

```
class Base(DeclarativeBase):
pass
class User(Base):
    __tablename__ = "user"
    id: Mapped[int] =
mapped_column(primary_key=True)
    name: Mapped[str]
engine = create_engine(
"sqlite:///app.db")
Base.metadata.create_all(
engine)
```

Insert and read back

```
with Session(engine) as s:
    s.add(User(name="ann"))
    s.commit()
    for u in
s.scalars(select(User)):
        print(u.id, u.name)
```

Users with post counts

```
stmt = (select(User.name,
func.count(Post.id))
    .join(User.posts,
isouter=True)
    .group_by(User.id))
rows =
session.execute(stmt).all()
```

Session per request

```
SessionLocal =
sessionmaker(engine)
def get_db():
    with SessionLocal() as s:
        yield s
```