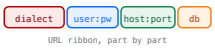
SQLAlchemy

A quick reference for SQLAlchemy, Python's SQL toolkit and ORM, from raw queries and connections to declarative models and relationships.

Engine & Connection URLs

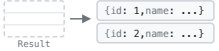
One create_engine call builds a pooled gateway to any database from a URL.

PURPOSE	COMMAND	RESULT
Build an engine (SQLite file).	`engine = create_engine("sqlite:///app.db")`	
Connect to PostgreSQL.	`create_engine("postgresql+psycopg:// user:pw@localhost:5432/mydb")`	
In-memory DB for tests.	`create_engine("sqlite://")`	
See the SQL it emits.	`create_engine(url, echo=True)`	
Build a URL safely.	`URL.create("postgresql+psycopg", username="u", ...)`	
Open a connection block.	`with engine.connect() as conn: ...`	

create_engine(url) builds one lazy, pooled Engine; open short with engine.connect() blocks that return the connection on exit

Raw SQL with text()

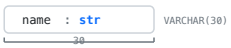
Run any SQL string with bound parameters, never string formatting.

PURPOSE	COMMAND	RESULT
Execute a query.	`conn.execute(text("SELECT * FROM users"))`	
Bind parameters (no f-strings).	`conn.execute(text("SELECT * FROM users WHERE id =:id"), {"id": 7})`	
One scalar value.	`conn.execute(text("SELECT count(*) FROM users")).scalar_one()`	
Rows as dicts.	`conn.execute(stmt).mappings().all()`	
Insert many rows at once.	`conn.execute(text("INSERT INTO t VALUES (:x)"), [{"x": 1}, {"x": 2}])`	
Commit the transaction.	`with engine.begin() as conn: ...`	

Wrap raw SQL in text() and pass :name placeholders with a params dict; engine.begin() commits on clean exit, rolls back on error

ORM Models

Map a Python class to a table with Mapped and mapped_column.

PURPOSE	COMMAND	RESULT
Declare a base class.	`class Base(DeclarativeBase): pass`	
Map a class to a table.	`classUser(Base):` `__tablename__ = "user_account"`	
Typed primary-key column.	`id: Mapped[int] =mapped_column(` `primary_key=True)`	
Typed column with length.	`name: Mapped[str] =mapped_column(String(30))`	
Nullable column (Optional).	`fullname: Mapped[str | None]`	
Create all tables.	`Base.metadata.create_all(engine)`	

2.0 style: subclass DeclarativeBase, set __tablename__, declare each column as Mapped[...] + mapped_column(); the Python type drives the SQL type

Session & Unit of Work

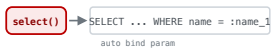
A Session collects changes; commit flushes them in one transaction.

PURPOSE	COMMAND	RESULT
Open a session.	`with Session(engine) as session:`	
A reusable session factory.	`Session = sessionmaker(engine)`	
Stage one new object.	`session.add(user)`	
Stage several at once.	`session.add_all([u1, u2])`	
Commit the unit of work.	`session.commit()`	
Undo on error.	`session.rollback()`	

add / add_all stage objects as pending; nothing hits the database until commit() flushes the whole batch in one transaction

Querying with select()

Build a SELECT as an object, then run it through the Session.

PURPOSE	COMMAND	RESULT
Build a SELECT.	`stmt = select(User).where(User.name == "ada")`	
Get model objects back.	`session.scalars(stmt).all()`	
Exactly one (or error).	`session.scalars(stmt).one()`	
Order and limit.	`select(User).order_by(User.name).limit(10)`	
Look up by primary key.	`session.get(User, 7)`	
Aggregate / count.	`session.scalar(select(func.count()).select_from(User))`	

select() builds the statement; scalars() returns mapped objects, get() hits the identity-map cache, one() raises if not exactly one

Insert, Update, Delete

Mutate through the Session (ORM) or with Core DML statements.

PURPOSE	COMMAND	RESULT
Insert (ORM).	`session.add(User(name="ada")); .commit()`	
Update (ORM, just assign).	`user.fullname = "Ada L."; session.commit()`	
Delete (ORM).	`session.delete(user); session.commit()`	
Bulk update (Core).	`update(User).where(User.id == 7) .values(name="x")`	
Bulk delete (Core).	`delete(User).where(User.name == "spam")`	
Run the Core statement.	`session.execute(stmt); session.commit()`	

ORM: add / assign / delete then commit; Core update()/delete() compile to one bulk statement, run via session.execute then commit

Relationships & Joins

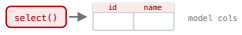
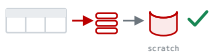
Link tables with relationship(); traverse or join across them.

PURPOSE	COMMAND	RESULT
Foreign-key column.	`user_id: Mapped[int] = mapped_column(ForeignKey("user_account.id"))`	
One-to-many relationship.	`addresses: Mapped[list["Address"]] = relationship(back_populates="user")`	
Many-to-one back-reference.	`user: Mapped["User"] = relationship(back_populates="addresses")`	
Traverse in Python.	`user.addresses.append(Address(email="a@x.com"))`	
JOIN two tables.	`select(User.name, Address.email).join(User.addresses)`	
Avoid the N+1 problem.	`select(User).options(selectinload(User.addresses))`	

ForeignKey ties rows in the DB; relationship() exposes the link in Python; join() and selectinload() pull related data without the N+1 storm

Read straight into pandas

Hand pandas a SQLAlchemy connection and a statement; get a DataFrame.

PURPOSE	COMMAND	RESULT
Read a SQL string.	`pd.read_sql(text("SELECT * FROM ..."), conn)`	
Read a select() statement.	`pd.read_sql_query(select(User.id, User.name), conn)`	
Read a whole table.	`pd.read_sql_table("user_account", engine)`	
Pass params safely.	`pd.read_sql(text("... WHERE id = :id"), conn, params={"id": 7})`	
Read in chunks (big tables).	`pd.read_sql(stmt, conn,chunksize=10_000)`	
Write a DataFrame back.	`df.to_sql("scratch", engine, if_exists="replace")`	

SQLAlchemy owns the connection and SQL, pandas owns the table: read_sql/read_sql_query/read_sql_table in, to_sql out