ORM & SQLAlchemy

Topics Covered

- ORM fundamentals & use cases
- SQLAlchemy architecture (Engine, Session, DBAPI)
- SQLAlchemy Core vs ORM
- MetaData, schema management & reflection
- Inspector vs MetaData

ORM Concepts & Architecture

ORM (Object Relational Mapping)

What is ORM?

- Bridging the gap between **Python objects** and **relational databases**.

- ORM maps **tables** → **classes**
- Rows → objects
- Columns → attributes

Why ORM is Used

- Faster development (less SQL writing)
- Cleaner, maintainable code
- DB-agnostic (Postgres, MySQL, SQLite)
- Built-in security (SQL injection protection)

SQL vs ORM

SQL	ORM
Full control	Faster dev
Verbose	Cleaner code
DB-specific	DB-independent
Harder to maintain	Easier to refactor

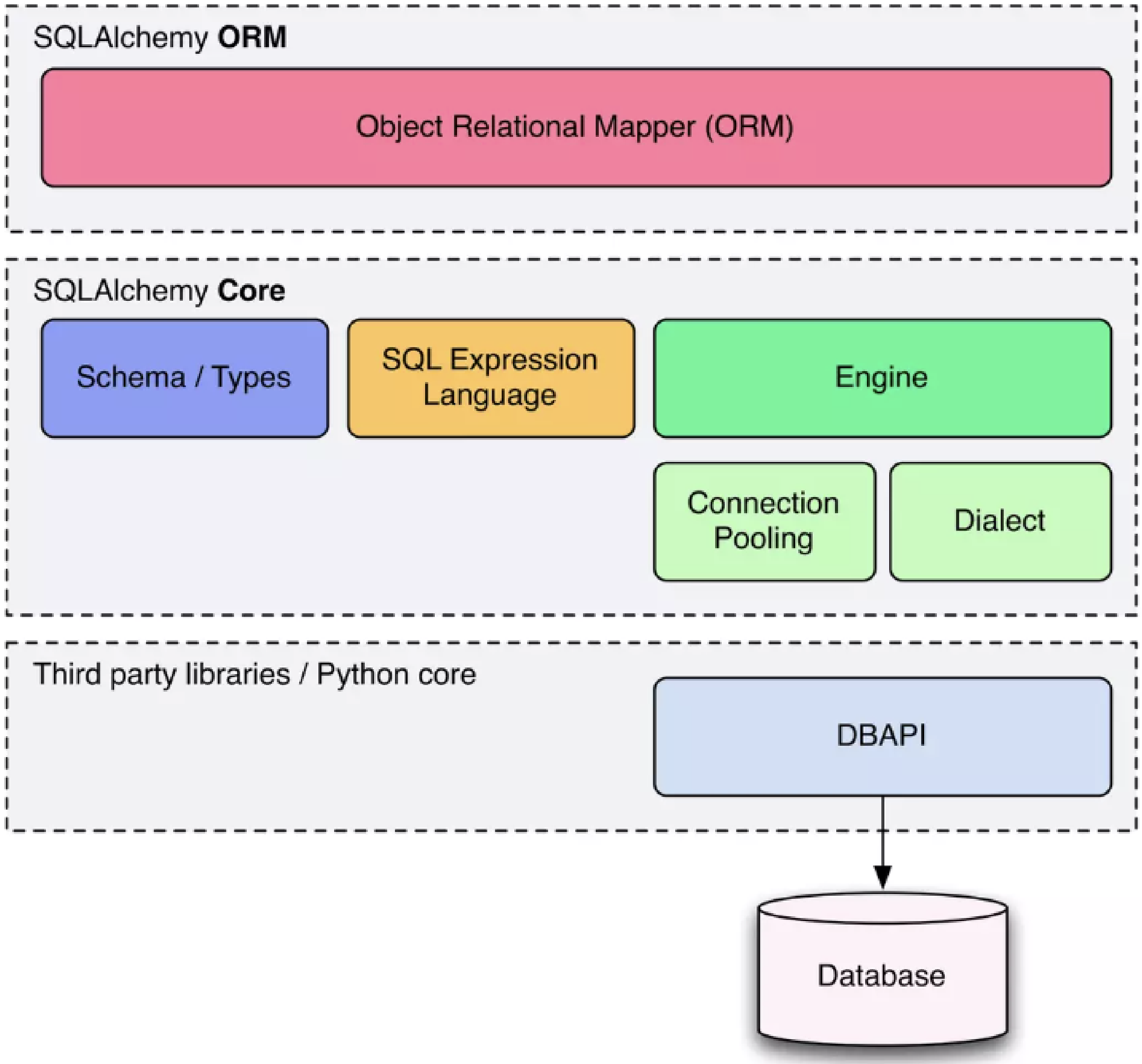
SQLAlchemy

- **SQLAlchemy** is a **Python SQL toolkit + ORM** that lets you work with databases using **Python code instead of raw SQL**.
- **Key Points**
  - Acts as a **bridge** between Python objects and relational databases
  - Supports **ORM + Core (SQL Expression Language)**
  - Works with PostgreSQL, MySQL, SQLite, Oracle, MSSQL
  - Gives you **ORM convenience** without hiding SQL completely

SQLAlchemy Philosophies

- **Explicit over implicit**
  You see what SQL is being generated. No magic.
- **SQL is not the enemy**
  SQLAlchemy respects SQL — it doesn’t try to hide it.
- **Separation of concerns**
  Engine (DB) ≠ Session (transactions) ≠ Models (business logic)
- **One ORM, many databases**
  Same code, different DBs. Minimal pain.
- **Control is optional, not forbidden**
  Start simple with ORM → drop to raw SQL when needed.

Architecture



SQLAlchemy **ORM**

Object Relational Mapper (ORM)

SQLAlchemy **Core**

Schema / Types

SQL Expression Language

Engine

Connection Pooling

Dialect

Third party libraries / Python core

DBAPI

Database

SQLAlchemy – Core

SQLAlchemy Core is the **low-level foundation** that deals directly with databases and SQL.

- **Engine**
  Entry point to the database. Manages DB connectivity.
- **Dialect**
  Translates generic SQLAlchemy commands into DB-specific SQL
  (Postgres, MySQL, SQLite, etc.)
- **Connection Pool**
  Maintains reusable DB connections for performance.
- **SQL Expression Language**
  Write SQL using Python expressions (SELECT, INSERT, JOIN).
- **Schema / Types**
  Python objects represent tables, columns, and datatypes.

SQLAlchemy – ORM

SQLAlchemy ORM sits on top of the Core and provides an **object-centric view**.

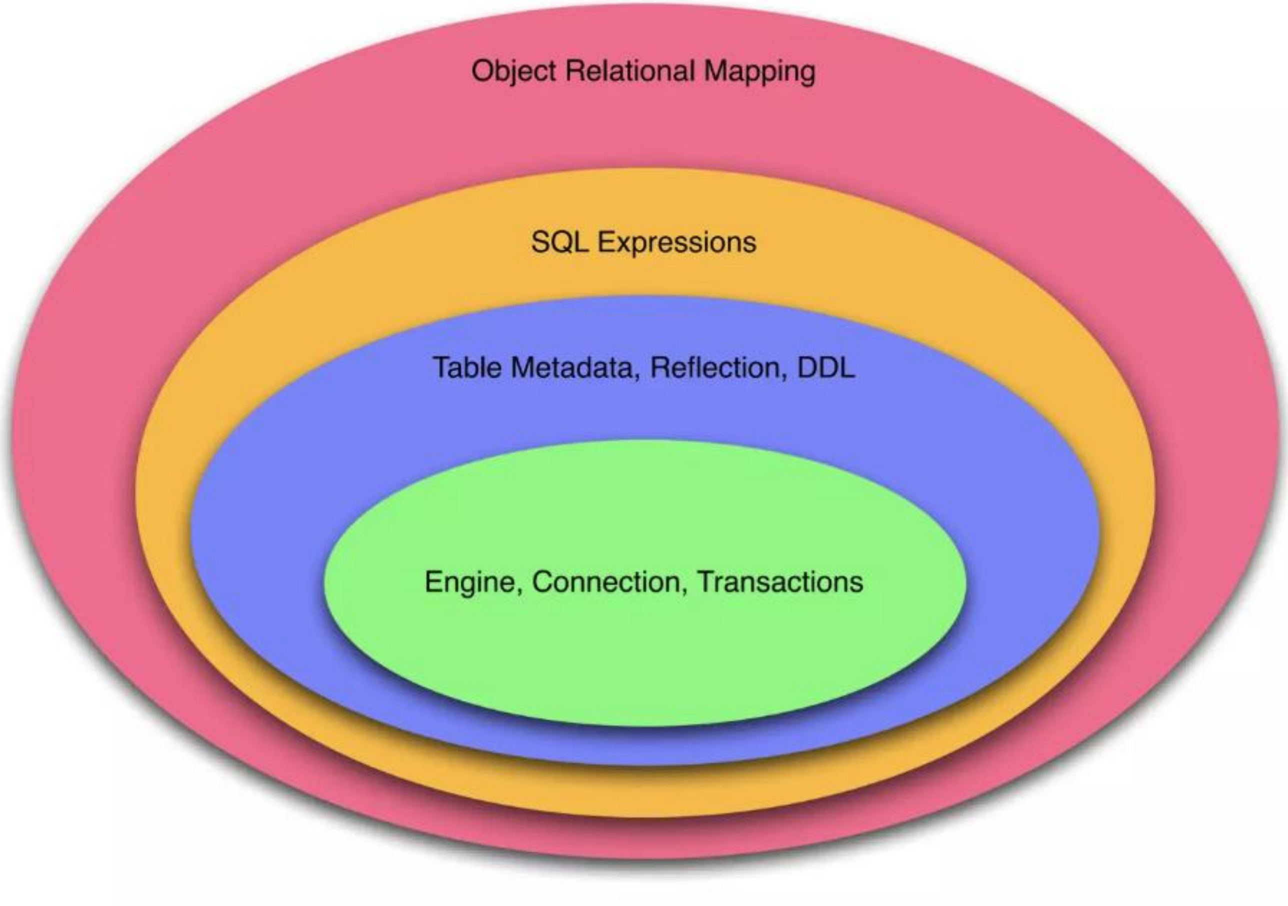
- Maps **Python classes** → **database tables**
- Maps **objects** → **rows**
- Handles persistence using the **Unit of Work pattern**
- Automatically generates SQL from mappings
- Uses **Core internally** to talk to the database

Perspective shift:

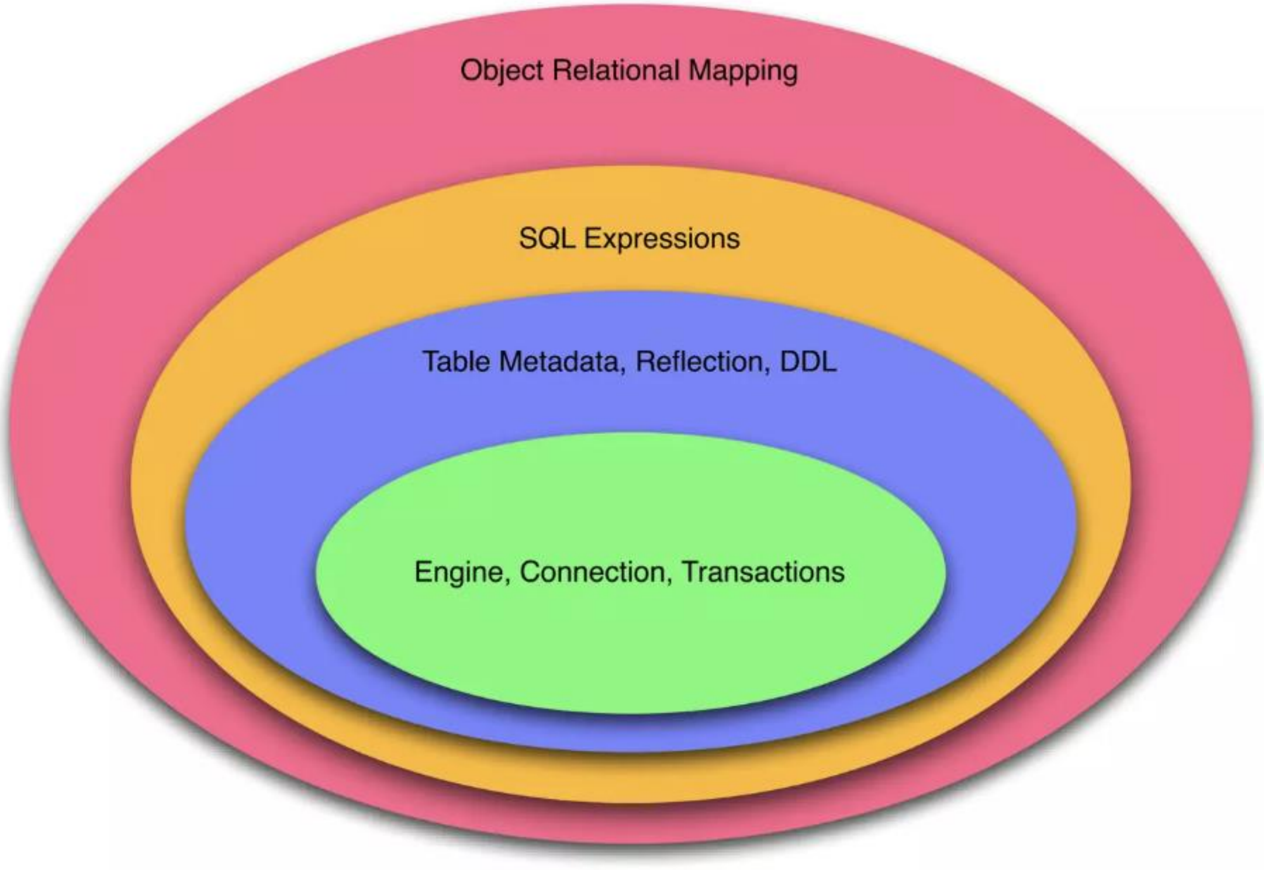
- Core → schema centric
- ORM → object centric

SqlAlchemy is like Onion

Can be learned from the inside out, or outside in


Object Relational Mapping
SQL Expressions
Table Metadata, Reflection, DDL
Engine, Connection, Transactions

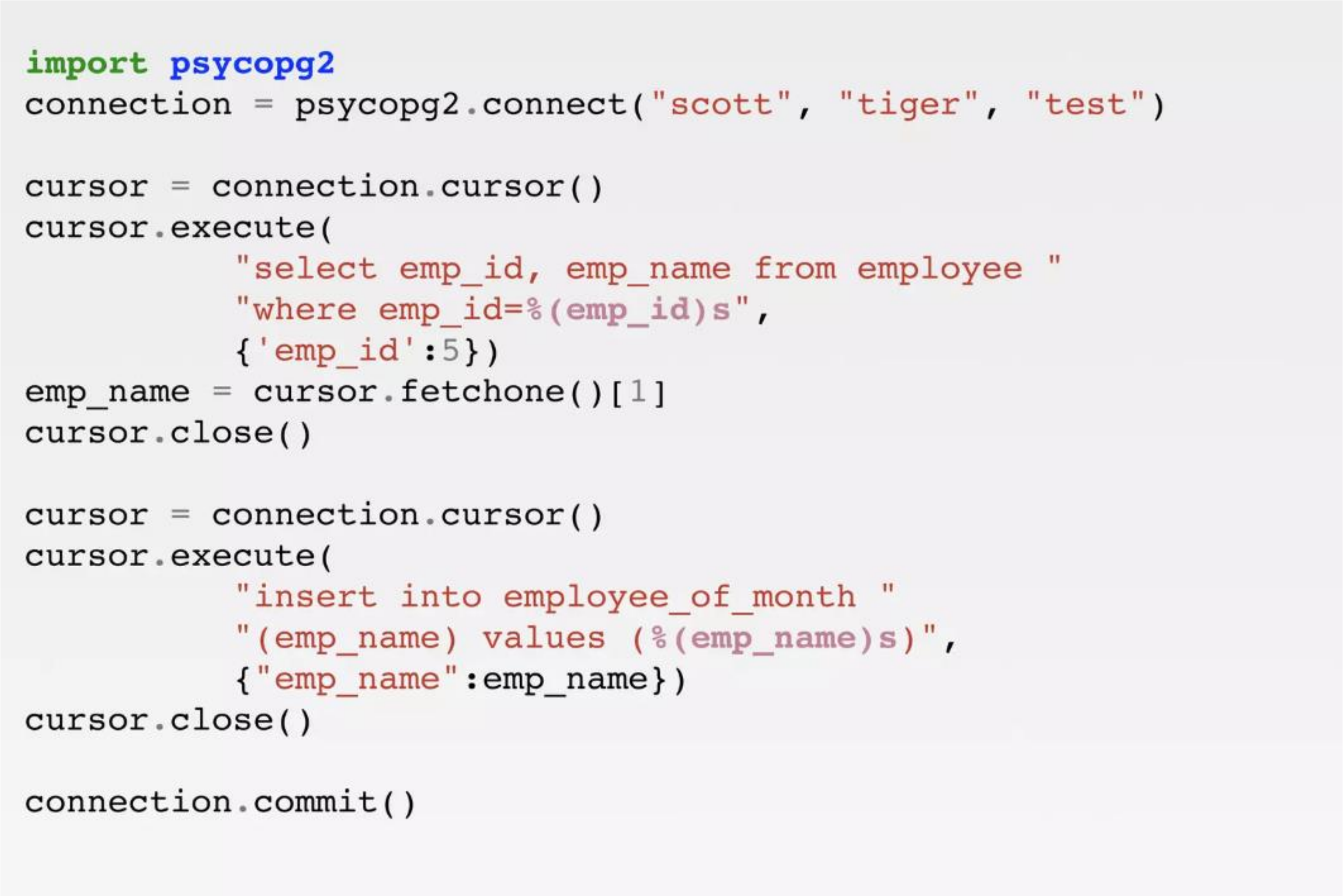
Level 1: Engines, connections, transactions


Object Relational Mapping
SQL Expressions
Table Metadata, Reflection, DDL
Engine, Connection, Transactions

Python DBAPI (PEP 249)

- **Python DBAPI** is a **standard specification** for how Python talks to relational databases.
- **What it Defines**
  - How to **connect** to a database
  - How to **execute SQL**
  - How to **fetch results**
- How **transactions** work (commit / rollback)
- **Common DBAPI Drivers**
  - psycopg2 → PostgreSQL
  - mysqlclient → MySQL
  - sqlite3 → SQLite
  - cx_Oracle → Oracle

DBAPI Example



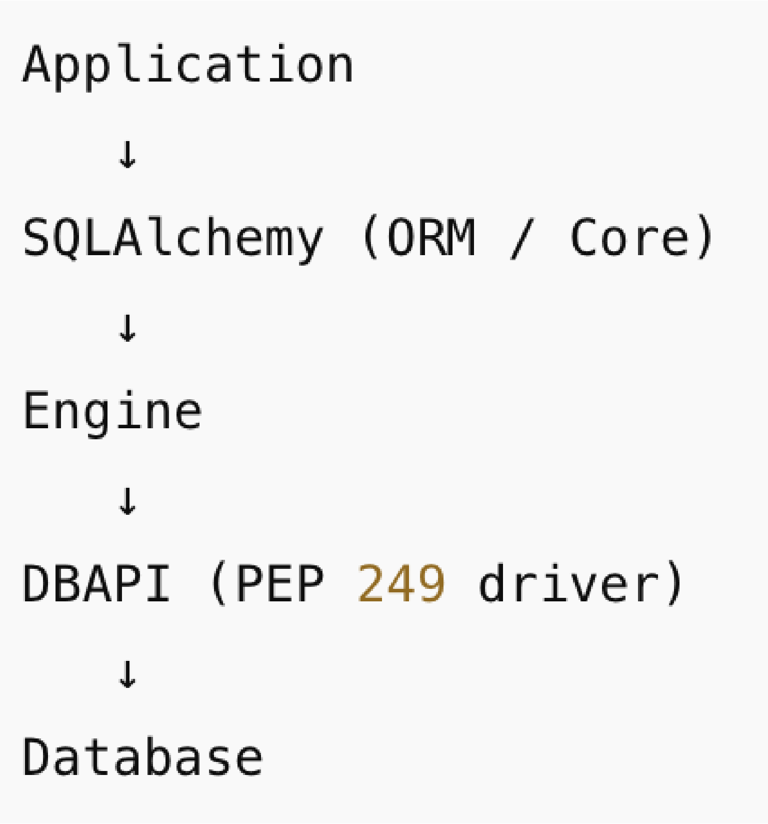
```python
import psycopg2
connection = psycopg2.connect("scott", "tiger", "test")

cursor = connection.cursor()
cursor.execute(
          "select emp_id, emp_name from employee "
          "where emp_id=%(emp_id)s",
          {'emp_id':5})
emp_name = cursor.fetchone()[1]
cursor.close()

cursor = connection.cursor()
cursor.execute(
          "insert into employee_of_month "
          "(emp_name) values (%(emp_name)s)",
          {"emp_name":emp_name})
cursor.close()

connection.commit()
```

SQLAlchemy Engine & DBAPI



```
Application
    ↓
SQLAlchemy (ORM / Core)
    ↓
Engine
    ↓
DBAPI (PEP 249 driver)
    ↓
Database
```

Engine is the **core interface** between SQLAlchemy and the database.

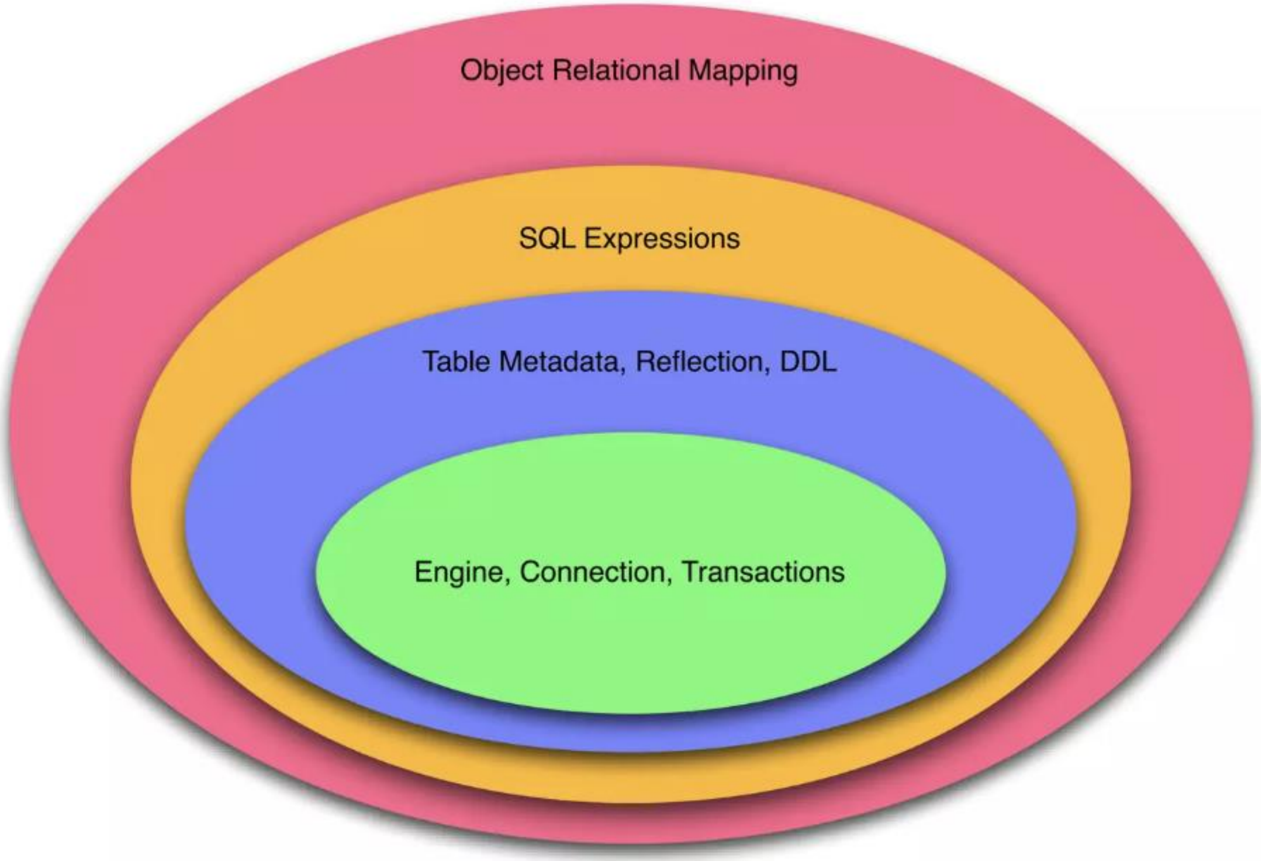
What the Engine Does

- Manages **database connections**
- Uses **connection pooling**
- Talks to the database via **DBAPI drivers**
- Executes SQL (generated by Core or ORM)

Role of DBAPI

- DBAPI is the **actual driver** (psycopg2, mysqlclient, sqlite3)
- Engine **does not speak DB protocol**
- Engine delegates low-level work to DBAPI

Level 2: MetaData, Tables


Object Relational Mapping
SQL Expressions
Table Metadata, Reflection, DDL
Engine, Connection, Transactions

MetaData (More Than Just Tables)

MetaData is the **central registry** of your database schema in SQLAlchemy.

- **What MetaData Actually Stores**
    - All **Table** objects
    - Column definitions & types
    - **Primary keys, foreign keys**
    - Constraints & indexes
    - Naming conventions

What is MetaData used for?

MetaData is **SQLAlchemy’s notebook** where it writes down:

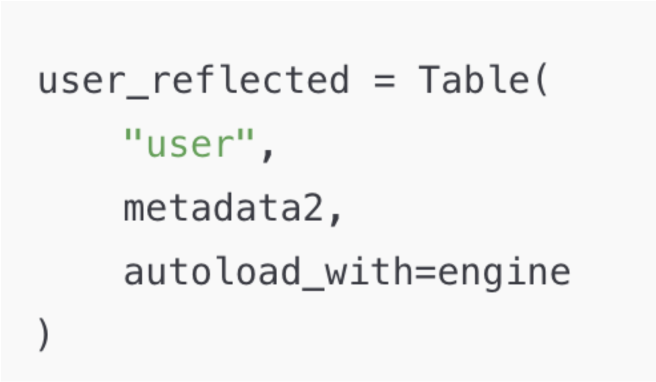
- What tables exist
- What columns they have
- How tables are connected

MetaData

- **MetaData is NOT a Database**
  - It does **nothing by itself**
  - No tables are created until you call: metadata.create_all(engine)
- **MetaData Knows Table Order**
  - SQLAlchemy figures out **dependency order**
  - Foreign keys are created safely
  - Drop order is also handled automatically
- **Reflection is MetaData's Superpower**
  - Loads schema from an **existing DB**
  - Zero Table definitions required
  - Table("user", metadata, autoload_with=engine)

Multiple MetaData Objects (Why & When)

- **Why create 2 MetaData objects?**
  - One for **tables you define**
  - One for **tables that already exist**
  - metadata = MetaData()   # tables we define
  - metadata2 = MetaData()   # tables we reflect
- **Reflection (reading existing DB schema)**
  - No CREATE TABLE needed
  - SQLAlchemy reads structure from DB
  - Useful for **legacy databases**
- **Drop ALL tables (dangerous but powerful)**
  - metadata.drop_all(engine)



```
user_reflected = Table(
    "user",
    metadata2,
    autoload_with=engine
)
```

Inspector (Database X-Ray Tool)

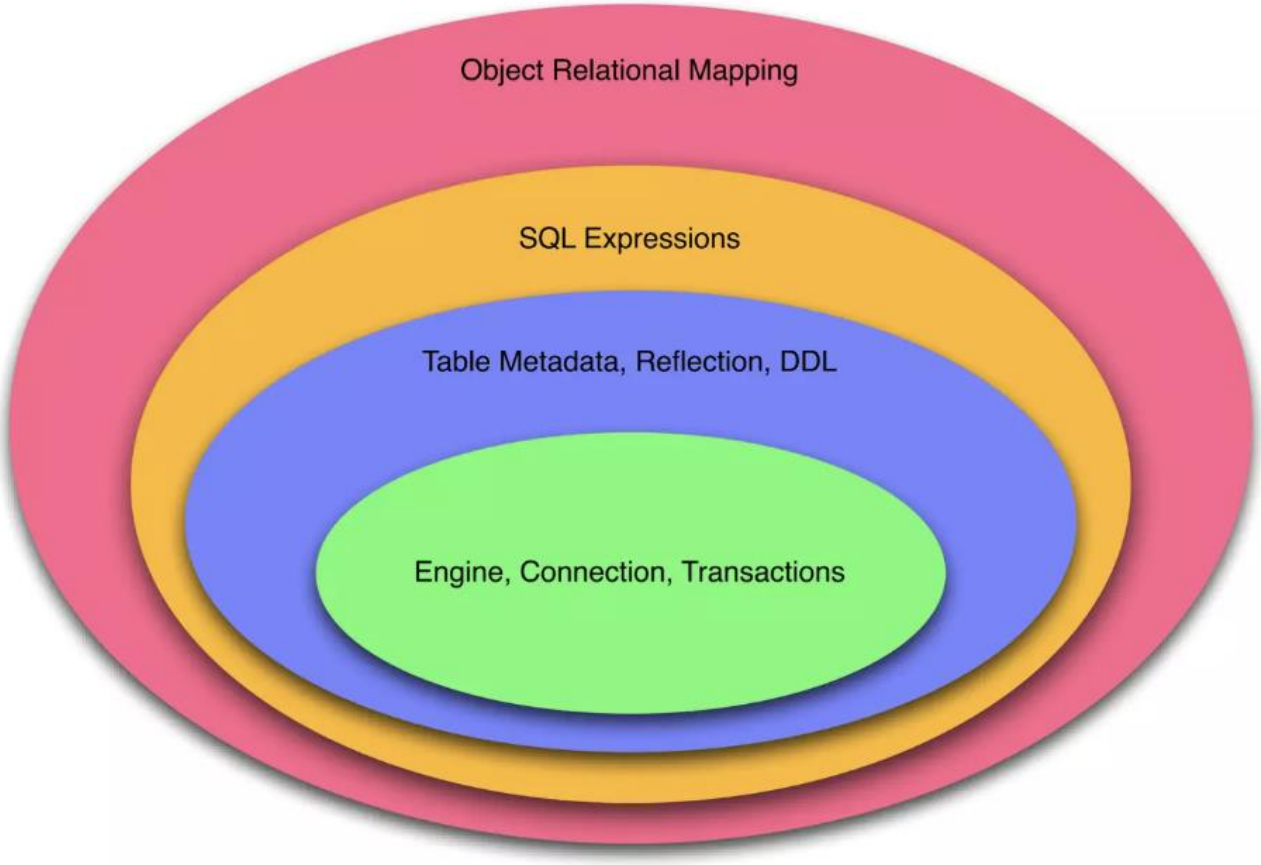
- **Inspector** lets you **ask questions about the database**.
  - from sqlalchemy import inspect
  - inspector = inspect(engine)
  - ➡️ List all tables -- inspector.get_table_names()
  - ➡️ See column names, types, nullability - inspector.get_foreign_keys("address")
- **When Inspector is used**
  - Debugging schema issues
  - Reverse-engineering databases
  - Writing migration tools

Why Inspector alone is NOT enough

- **Inspector can only:**
  - List tables
  - Show columns
  - Show foreign keys
- You **cannot**:
  - Build SQL expressions
  - Join tables
  - Insert / update rows
  - Use ORM
  - Run migrations
- Inspector is **read-only**.

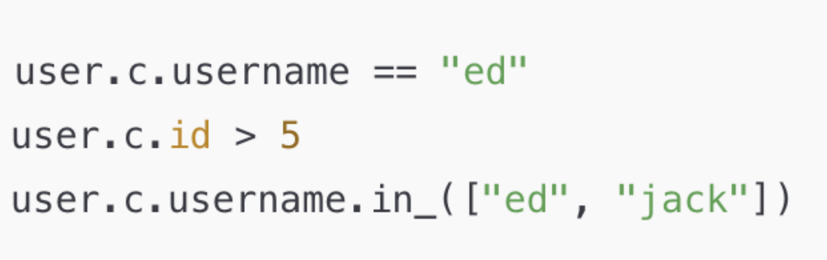
- With reflected MetaData
  - user = Table("user", metadata, autoload_with=engine)
- You can now:
  - select(user).where(user.c.id == 1)
- Write queries
- Use Core / ORM
- Participate in transactions
- Generate migrations

Level 3: SQL Expressions


Object Relational Mapping
SQL Expressions
Table Metadata, Reflection, DDL
Engine, Connection, Transactions

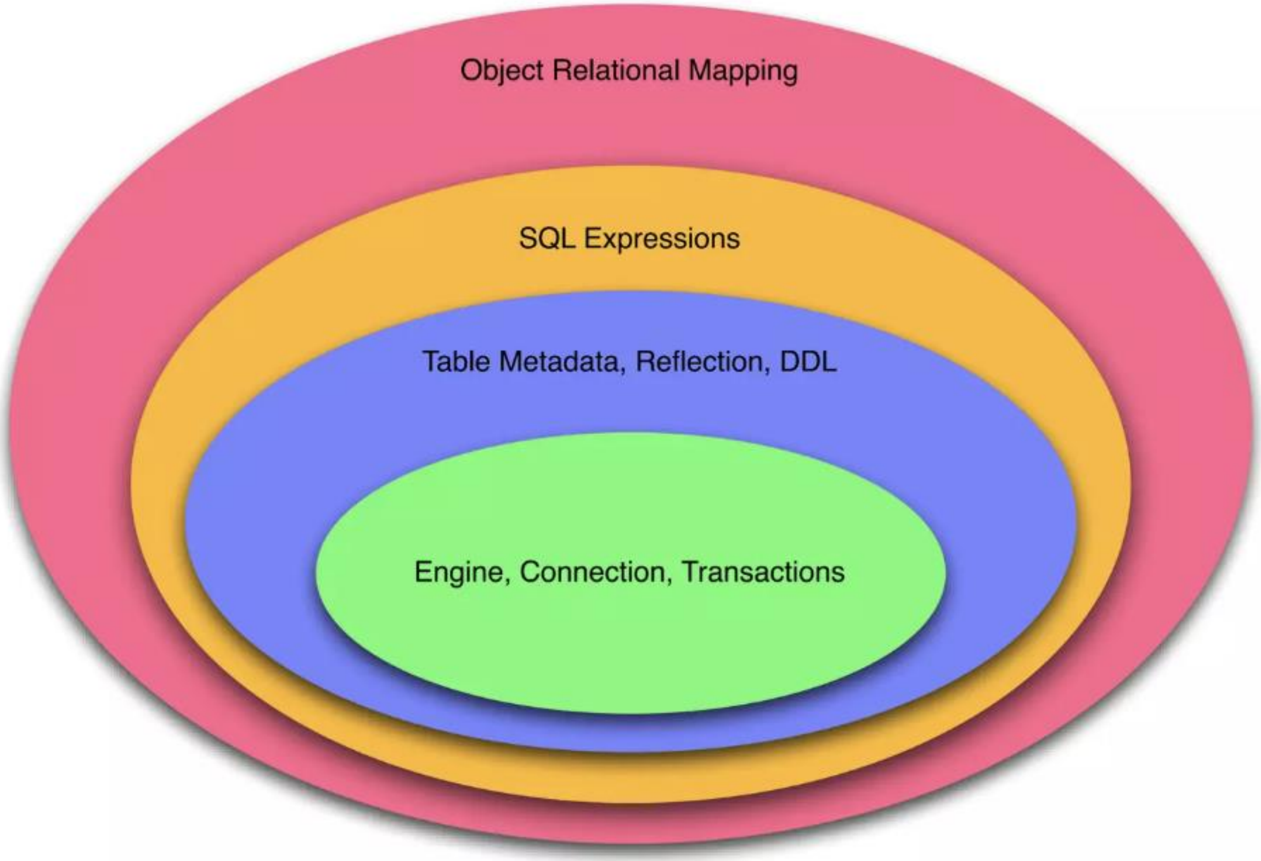
SQLAlchemy Core – SQL Expression Language

- Pythonic way to **build SQL** using objects
- Produces **database-agnostic SQL**
- Foundation for both **Core and ORM**
- **Key Ideas**
  - Table.c.column → column reference
  - == , > , in_() → SQL expressions
  - & , | , and_() , or_() → AND / OR
  - Expressions compile differently per DB dialect
- **Transactions (2.x rule)**
  - All INSERT / UPDATE / DELETE must be inside: **with engine.begin():**



```
user.c.username == "ed"
user.c.id > 5
user.c.username.in_(["ed", "jack"])
```

Level 4:
ORM


Object Relational Mapping
SQL Expressions
Table Metadata, Reflection, DDL
Engine, Connection, Transactions

ORM (Object Relational Mapping)

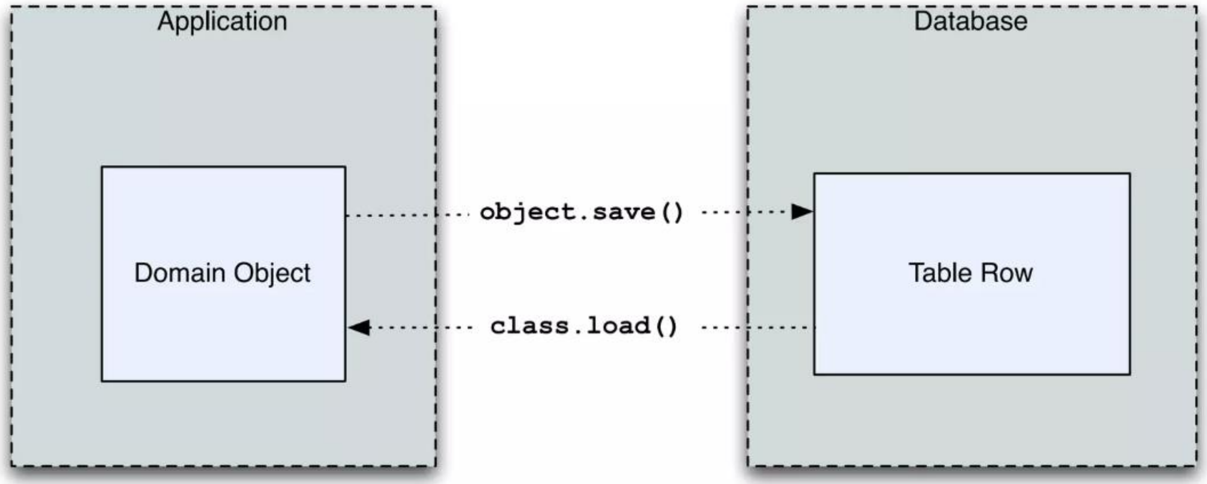
- **ORM = Bridge between Objects and Tables**
  - Maps **classes** → **tables**
  - Maps **objects** → **rows**
  - Maps **attributes** → **columns**
- You work in **code**, ORM talks **SQL**.

Why ORM Exists

- Writing raw SQL everywhere is:
  - repetitive
  - error-prone
  - hard to maintain
- Applications think in **objects**, DB thinks in **tables**
- ORM solves the **object–table mismatch**.

What ORM Actually Does

The most basic task is to translate between a domain object and a table row.



- Converts Python objects to SQL
- Executes SQL via DB drivers
- Converts result rows back to objects
- Manages transactions & sessions
- **Important:**
  ORM does **not** replace SQL — it generates it.

Core Components of an ORM

- **Model** → represents a table
- **Session / Unit of Work** → manages transactions
- **Mapper** → maps class ↔ table
- **Query Builder** → builds SQL
- These pieces work together — not magic.