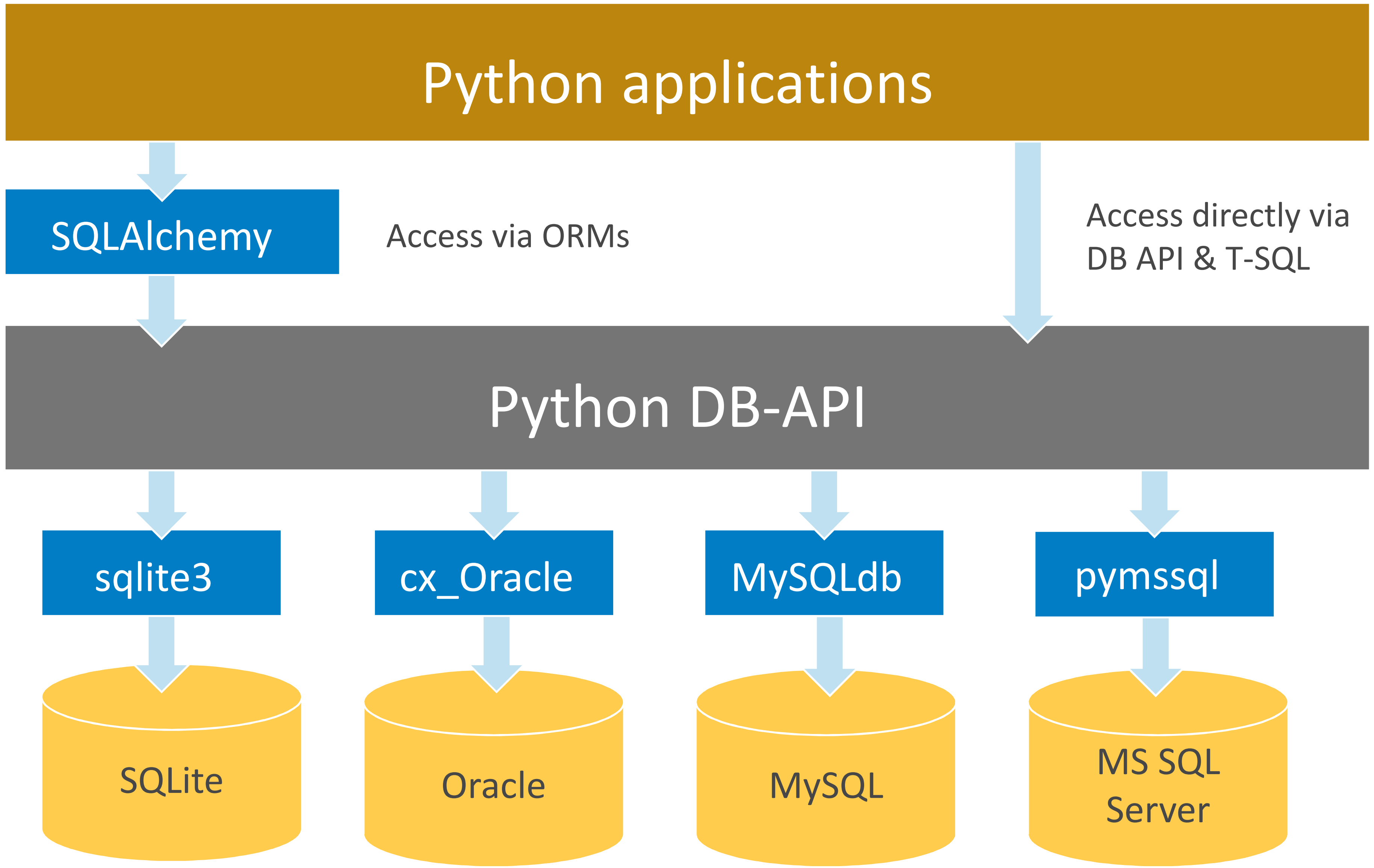
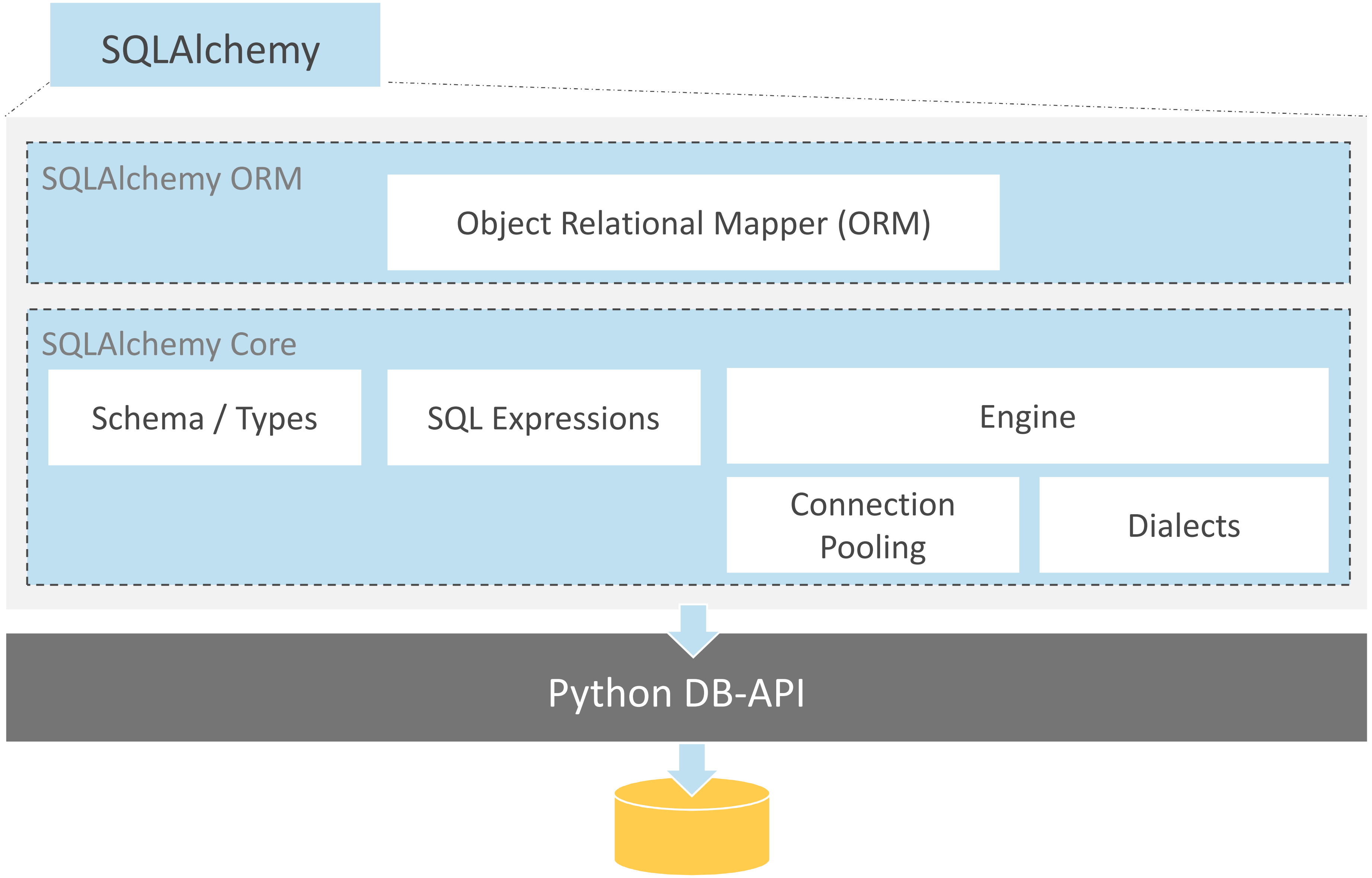
# SQLAlchemy ORM

- Objectives
  - Learn what ORMs have to offer over the basic DB-API
  - Use SQLAlchemy’s ORM to work with data at the object level
  - Define schemas with the ORM model
  - Map custom classes to DB tables via the ORM
  - Create relationships between related model objects
  - Work with DB transactions
  - Use lazy and eager loading to enhance performance

# Recall [DB-API Architecture]



# SQLAlchemy Architecture



# SQLAlchemy ORM [getting started]

- SQLAlchemy ORM builds on Core
  - so we use many of the same foundational objects

```
# same as before
from sqlalchemy import create_engine

memory_db = 'sqlite:///:memory:'

# use one engine instance per DB URL.
engine = create_engine(memory_db, echo=True)
```

# SQLAlchemy ORM [declaring your schema]

We need a single common **Base** class (created via a factory method)

```
import sqlalchemy
import sqlalchemy.ext.declarative
from sqlalchemy import Column, Integer, String

# use one base instance per DB / model hierarchy.
Base = sqlalchemy.ext.declarative.declarative_base()

class User( Base ):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

class Address( Base ):
    __tablename__ = 'addresses'
    id = Column(Integer, primary_key=True)
    email_address = Column(String)
    user_id = Column(Integer)
```

# SQLAlchemy ORM [creating the DB tables]

```
import sqlalchemy

# single Base class from before:
Base.metadata.create_all(engine)
```

After declaring the models via `class ModelClass( Base )`, we create tables via **metadata** registry.

## SQLAlchemy ORM [creating objects]

Creating ORM objects is just like regular objects

```
u = User(
   name='Jeff',
   fullname='Jeff Thompson',
   password='123')

print( u.name )
# prints: Jeff

print( u.id )
# prints: None
```

Initially, the primary key is None.

## SQLAlchemy ORM [unit of work]

- SQLAlchemy uses the unit of work pattern
  - managed through a Session object

```
from sqlalchemy.orm import sessionmaker

# one session_factory per engine
session_factory = sessionmaker(bind=engine)

session = session_factory()
# work with session object
```

Each unit of work is initiated by creating a session object.

# SQLAlchemy ORM [inserting objects]

1. Create a new object and set values.

2. Create a session.

3. Add the object

4. Commit all changes when ready.

```python
u = User(
   name='Jeff',
   fullname='Jeff Thompson',
   password='123')

session = session_factory()

session.add( u )
session.commit()
```

Make as many changes as you wish prior to commit.
Some operations will flush pending items to the DB.

# SQLAlchemy ORM [querying data]

Create a query based on a model

filter maps to **where** in SQL.

Can optionally order by columns.

```
session = session_factory()

users = session.query( User ). \
             filter(User.name=='Jeff'). \
             order_by(User.created.desc())

for user in users:
     print("Email {0} at {1}.".format(
          user.name, user.email_address
     )
```

# SQLAlchemy ORM [querying data]

- Reading data can be done via
  - iterating the result set
  - calling `results.all()`
    - returns a list
  - calling `results.one()`
    - returns the single item or raises an error
  - Call results.count()
    - returns the number of items in the query

## SQLAlchemy ORM [filtering data]

| SQL OPERATION | Example in SQLAlchemy |
|---|---|
| equals | `query.filter(User.name == 'ed')` |
| not equals | `query.filter(User.name != 'ed')` |
| LIKE | `query.filter(User.name.like('%ed%'))` |
| IN (values) | `query.filter(User.name.in_(['ed', 'wendy', 'jack']))` |
| NOT IN | `query.filter(~User.name.in_(['ed', 'wendy', 'jack']))` |
| IS NULL | `query.filter(User.name == None)` |
| IS NOT NULL | `query.filter(User.name != None)` |
| AND | `query.filter(and_(`<br>`       User.name == 'ed', User.fullname == 'Ed Jones'))` |
| AND (implicit) | `query.filter(User.name == 'ed')`<br>`     .filter(User.fullname == 'Ed Jones')` |
| OR | `query.filter(or_(User.name == 'ed', User.name == 'wendy'))` |

## SQLAlchemy ORM [transactions]

- Session manages the transaction

On success, user 42 will be updated.

```
session = session_factory()

try:
    user = session.query(User).filter(User.id==42).one()
    user.visit_count += 1
    call_other_method_which_may_fail()
    session.commit()
except:
    session.rollback()
```

On error, user 42 will **not** be updated.

# SQLAlchemy ORM [relationships]

ForeignKey controls DB constraints at the ORM level.

```
from sqlalchemy.orm import relationship, backref

class User( Base ):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)

    addresses = relationship("Address",
         order_by="Address.id", backref="user")

class Address(Base):
    __tablename__ = 'addresses'
    id = Column(Integer, primary_key=True)
    email_address = Column(String)
    user_id = Column(Integer, ForeignKey('users.id'))
```

relationship links in-memory models together via DB foreign keys.

## SQLAlchemy ORM [relationships and objects]

```
session = session_factory()

tom = session.query(User) \
             .filter(User.name == 'Tom') \
             .one()

tom.addresses.append(Address(email_address='tom@yahoo.com'))

session.commit()
```

Modifying the related collection will implicitly insert objects in the Addresses table.

# SQLAlchemy ORM [lazy loading]

Initial query returns user tom with a SQL query.

```
session = session_factory()

tom = session.query(User) \
             .filter(User.name == 'Tom') \
             .one()

# SQL emitted to the server:
# SELECT * FROM users WHERE name = ?

address_count = len(tom.addresses)

# SQL emitted to the server:
# SELECT * FROM addresses WHERE ? = user_id
```

Accessing a navigational field issues a second query to pull in all the addresses.

# SQLAlchemy ORM [eager loading]

```
session = session_factory()

tom = session.query(User)
           .options(joinedload(User.addresses)).\
           .filter(User.name == 'Tom')
           .one()

# SQL emitted to the server:

# SELECT * FROM users
# LEFT OUTER JOIN addresses ON users.id = addresses.user_id
# WHERE users.name = ? ORDER BY addresses_1.id

address_count = len(tom.addresses)

# SQL emitted to the server:
# none
```

Using **joinedload** option allows us to *eager load* addresses.

## SQLAlchemy ORM [deleting objects]

```
session = session_factory()

tom = session.query(User)
           .filter(User.name == 'Tom')
           .one()

session.delete( tom ) # leaves tom's addresses
```

↑

Without redefining our relationship, tom's addresses will remain.

## Summary

- ORMs allow us to work at a high level than pure SQL
- SQLAlchemy is the most popular ORM for Python
- The relationship object creates navigable relationships
- We perform CRUD with ORM
- SQLAlchemy always uses transactions
- Be aware of when lazy loading is used and add eager loading as needed