Chapter 15: SQLAlchemy Basics

Objectives

- Understand SQLAlchemy concepts
- Interface with a DBMS
- Create tables
- Understand the concept of models
- Create models
- Fetch results from queries
- Perform non-query statements

SQLAlchemy Overview

- Enterprise-level data persistence
- Use data abstractions, not the data itself
- Power and flexibility of SQL, but no SQL coding needed
- Simple Pythonic interface
- "relational algebra engine"
- Two parts:
  - Core: is SQL abstraction toolkit (schemas + types)
  - ORM: Object Relational Mapper (best-known part, but optional)
- Either part can be used independently

SQLAlchemy is a framework that provides data persistence at a high level. Your business logic involves data abstractions, not the data itself. The data is stored in a database, but you don't need to use SQL to access it.

Data access is accomplished by using table and other objects which act like normal Python objects.

There are two main parts. The core is an abstraction toolkit. It provides tools for working with schemas and types. The ORM is the object relational mapper. It is the best known part, but you do not necessarily need to use it.

Creating an engine

- Use create_engine()
- Different syntax for different DBMS

To get started using SQLAlchemy, create a database engine with the create_engine() function. This provides access to a database which will store the data.

An engine corresponds to a database server.

The create_engine() function requires a parameter that follows this pattern:

```
dialect+driver://user:password@host:port/dbname
```

"dialect" is the name of a SQLAlchemy dialect (e.g. SQLite, mysql, PostgreSQL, oracle, mssql, and firebird), and "driver" is the name of a Python DB API module such as pyodbc or mysqldb. If a driver not specified, a default DBAPI module will be imported.

Different databases (i.e., different engines) may require different parameters to create_engine(). See the table on the next page for examples.

For verbose output of SQLAlchemy statements, add the named parameter Echo with a value of True. This will log all statements, with a repr() of their parameter lists, to the DB engine's logger, which defaults to STDOUT. Logging can be toggled via the echo attribute of the engine. Note that at this point, the credentials are not checked; this will happen when you make a connection. This is just setting up a database server for use by specifying the credentials and other pertinent information. Once you have establish one or more connections, the engine will manage the pool of connections for you.

Example

sa_create_engine.py

```
#!/usr/bin/python

from sqlalchemy import create_engine

# for SQLite3:
# engine = create_engine('sqlite:///:memory', echo=True)

engine = create_engine(
    'mysql+pymysql://scripts:scripts@localhost/pyepa',
    echo=True
)

print(engine)
```

sa_create_engine.py

```
Engine(mysql+pymysql://scripts:***@localhost/pyepa)
```

NOTE	For a multiple-process application that uses the os.fork() system call, or for example the Python multiprocessing module, use a separate Engine for each process.

Table 21. Common create_engine parameter examples

DBMS	Parameter
MSSQL Server	mssql+pyodbc://dsn mssql+pyodbc://user:pw@host/database
MySQL	mysql://user:pw@host/database mysql+mysqldb://user:pw@host/database mysql+oursql://user:pw@host/database
Oracle	oracle+cx_oracle://user:pw@host:port/sidname
PostgreSQL	postgresql://user:pw@host:port/database postgresql+psycopg2:///user:pw@host/database
SQLite3 (memory)	sqlite://
SQLite3 (file)	sqlite:////absolute-path sqlite:///relative-path

Making the connection

- Connect to engine
- Connection object is used to query and update the DB

To make use of an engine, you need a connection to it. This is accomplished via the connect() method of the engine object returned by create_engine().

From the connection, you can create objects to perform selects or non-query statements such as inserts, updates, and deletes.

You can directly execute SQL statements from the connection,but it is usually better to create reusable query objects, or, for a higher level of abstraction, Table objects.

Example

sa_dbconnect.py

```
#!/usr/bin/python
from sqlalchemy import create_engine

engine = create_engine(
    'mysql+pymysql://scripts:scripts@localhost/pyepa',
    echo=False
)

conn = engine.connect()

s = conn.execute('select * from presidents where num = 16')

row = s.fetchone()

print(row.lname)
```

Mapping table objects

- Map to existing DBMS tables
- Metadata contains the schema details
- Set autoload to True
- Select from table

To use existing tables, create a Table object. This requires the table name, a Metadata object, and specify the autoload parameter with a value of True.

Use the MetaData class to create a metadata object which can be used for your database. The metadata is associated with a particular engine.

Once you have a table, you can use it both for querying and for returning results.

Example

sa_autoload_presidents.py

```
#!/usr/bin/python
from sqlalchemy import create_engine, select, MetaData, Table
from sqlalchemy.orm import sessionmaker

engine = create_engine(
    'sqlite:///../DATA/PRESIDENTS',
    echo=False
)

metadata = MetaData(bind=engine)

conn = engine.connect()

SESSION = sessionmaker(bind=engine)
session = SESSION()

pres = Table(
    'presidents',
    metadata,
    autoload=True,
)

q = session.query(pres).first()
print(q)
print(q.fname)
print(q.lname)

q = session.query(pres).filter_by(party='Republican').first()
print(q, '\n')

q = session.query(pres).all()

print(q[:3], '\n')  # first 3 rows

for row in q:
    print(row)

# q = session.query(customer_account).filter_by(cust_acct_id=6928638).first()
#
# print(q)
```

sa_autoload_presidents.py

```
(1, 'Washington', 'George', datetime.date(1789, 4, 30), datetime.date(1797, 3,
4), 'Westmoreland County', 'Virginia', datetime.date(1732, 2, 22),
datetime.date(1799, 12, 14), 'no party')
George
Washington
(16, 'Lincoln', 'Abraham', datetime.date(1861, 3, 4), datetime.date(1865, 4, 15),
'Hodgenville, Hardin County', 'Kentucky', datetime.date(1809, 2, 12),
datetime.date(1865, 4, 15), 'Republican')

[(1, 'Washington', 'George', datetime.date(1789, 4, 30), datetime.date(1797, 3,
4), 'Westmoreland County', 'Virginia', datetime.date(1732, 2, 22),
datetime.date(1799, 12, 14), 'no party'), (2, 'Adams', 'John',
datetime.date(1797, 3, 4), datetime.date(1801, 3, 4), 'Braintree, Norfolk',
'Massachusetts', datetime.date(1735, 10, 30), datetime.date(1826, 7, 4),
'Federalist'), (3, 'Jefferson', 'Thomas', datetime.date(1801, 3, 4),
datetime.date(1809, 3, 4), 'Albermarle County', 'Virginia', datetime.date(1743,
4, 13), datetime.date(1826, 7, 4), 'Democratic - Republican')]

(1, 'Washington', 'George', datetime.date(1789, 4, 30), datetime.date(1797, 3,
4), 'Westmoreland County', 'Virginia', datetime.date(1732, 2, 22),
datetime.date(1799, 12, 14), 'no party')
(2, 'Adams', 'John', datetime.date(1797, 3, 4), datetime.date(1801, 3, 4),
'Braintree, Norfolk', 'Massachusetts', datetime.date(1735, 10, 30),
datetime.date(1826, 7, 4), 'Federalist')
(3, 'Jefferson', 'Thomas', datetime.date(1801, 3, 4), datetime.date(1809, 3, 4),
'Albermarle County', 'Virginia', datetime.date(1743, 4, 13), datetime.date(1826,
7, 4), 'Democratic - Republican')
```

Flask and SQLAlchemy

Flask is not integrated with any database managers. The developer is free to use any database package, or implement a custom package.

Given that, most developers do use existing database tools.

For direct access to databases, the many modules that implement the DB API are available. Typically, you might abstract these databases into classes that hide the DB details from Flask.

You could also use the Flask-SQLAlchemy extension, and use ORM to create classes that represent database tables. In this case, SQLAlchemy will generate the actual SQL commands based on DB classes that you define. If you already have a database you want to use, SQLAlchemy can automatically create classes that map to the existing tables.

NOTE For NoSQL databases, the extensions Flask-CouchDB and Flask-PyMongo provide access,

Connecting to a database

- Typically connect in the main script or *init*.py
- Connection information from environment or config.py

A Flask app typically connects to the database at the main script level, maybe in *init*.py, depending on the project layout.

This connection can then be shared by different modules in the project.

You should not store sensitive information like passwords in source code, so that information can come from a separate file, user input, or a config module that is part of your project.

NOTE Flask is not integrated with any database managers. The developer is free to use any database package, or implement a custom package.

Creating a get_db() function

- Convenient to make a universal function
- Use the g (global) variable from the app context

For convenience, you can create a global function that will retrieve the database connection.

You might also create a function that executes a query and returns the result set as tuples, dictionaries, or whatever your app needs. It would combine the execute() and fetch() methods.

Using Flask-SQLAlchemy

- Create models (classes that map to DB tables)
- Generates SQL as needed
- Can create DB from scratch
- Interact with DB via objects, not SQL

SQL-Alchemy is a popular choice for supporting Flask apps. It is relatively easy to use, and works with SQLite3, MySQL, PostgreSQL, Oracle, Microsoft SQL Server, and others.

It works by using class-based *models*, which map to the database tables.

Creating models

- Create class that inherits from db.Model
- Add columns as class variables
- Add str() or repr() methods as desired
- Add constructor (*init*) to populate the model

To create a model in SQL Alchemy, define a class that inherits from the Model base class. To add columns, add class variables of type Column. The arguments to Column define the data type and other properties of the column

Add a *repr* method for displaying the raw model.

Creating the database

- Call create_all from the DB object

Once the models are defined, call create_all() to generate and execute the SQL to build the initial database.

Example

flask_sa_models.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_DATABASE_URI'] = 'oracle://FREDDY:L0LZ@(DESCRIP...)'
db = SQLAlchemy(app)

class President(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    firstname = db.Column(db.String(80))
    lastname = db.Column(db.String(80))
    birthstate = db.Column(db.String(2))

    def __init__(self, firstname, lastname, state):
        self.firstname = firstname
        self.lastname = lastname
        self.birthstate = state

    def __repr__(self):
        return '<President {} {}>'.format(self.firstname, self.lastname)

def create_db():
    '''Warning! Only run this once for production!!!!'''
    db.create_all()

if __name__ == '__main__':
    create_db()
```

Adding and accessing data

- To use the database, create instances of models
- Add model instances to db session with db.session.add()
- Call session.commit() to save all added instances

To add data to the database, create an instance of a model, and populate it with appropriate data. Add the instance to the database session. When one or more instances have been added, use db.session.commit() to save all instances added since the last commit.

Example

flask_sa_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
from flask import Flask
# from flask.ext.sqlalchemy import SQLAlchemy
from flask_sa_models import db, President

app = Flask(__name__)

@app.route('/president/<int:term>')
def add_pres(term):
    # get user input here from form or API or wherever....
    p = President('Abraham', 'Lincoln', 'IL')
    db.session.add(p)
    db.session.commit()

    return '''<h1>Added.</h1>''', 201


if __name__ == '__main__':
    app.run(debug=True)
```

Building and using queries

- Create select objects
- Equivalent to SQL 'select' statements
- Can select from more than one table
- Can add filters, ordering, etc.
- Call select() from table
- Call execute() from query
- Execute returns an iterable result

From a Table object, you can call the select() method to create a query object. This query can then be executed, via the aptly-named execute() method.

Fetching results

- All queries start with session
- Use query() to get a query object
- Call methods on query object

To get data from the database, use the session object call the **query()** method, passing in the name of the object.

From that object, you can call **all()**, **count()**, or other query methods.

Example

sa_movie_search_db.py

```
#!/usr/bin/python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from sa_movie_models import Movie, Director

def main():
    session = setup()
    do_query_1(session)
    print()
    do_query_2(session)
    session.close()

def setup():
    ENGINE = create_engine('mysql+pymysql://scripts:scripts@localhost/pyepa',
echo=False)
    SESSION = sessionmaker(bind=ENGINE)
    return SESSION()

def do_query_1(session):
    for director in session.query(Director):
        if director.first_name == 'Steven':
            director.first_name = 'Bob'
        print(director.first_name, director.last_name)
        for movie in director.movies:
            print("\t{0} ({1})".format(movie.movie_name, movie.year))

def do_query_2(session):
    movie_name_query = 'M*A*S*H'

    # select * from Movie where movie_name  = 'M*A*S*H'
    for result in session.query(Movie).filter(Movie.movie_name==movie_name_query
):
        print("{0} was directed by {1} {2}".format(
            movie_name_query,
            result.person.first_name,
            result.person.last_name,
        ))

if __name__ == '__main__':
    main()
```

Table 22. SQLAlchemy Query Functions

all()	Returns all the objects in the query
count()	Returns the total number of rows of a query
filter()	Filters the query by applying a criteria
delete()	Removes from the database the rows matched by a query
distinct()	Applies a distinct statement to a query
exists()	Adds an exists operator to a sub-query
first()	Returns the first row in a query
get()	Returns the row referenced by the primary key parameter passed as argument
join()	Creates a SQL join in a query
limit()	Limits the number of rows returned by a query
order_by()	Sets an order in the rows returned by a query

Chapter 15 Exercises

Exercise 15-1 (knight_alchemy.py)

Write a script to create a simple database (MySQL/MariaDB or SQLite3) to store information about the Knights of the Round Table.

If necessary, make sure you've installed SQLAlchemy:

```
pip3 install sqlalchemy
```

Or if you're using Anaconda:

```
conda install sqlalchemy
```

In this script we'll need to:

Create a connection to your preferred database, along with a SQLAlchemy Session and Base from declarative_base()

Create a Knight class that extends from the SQLAlchemy Declared Base with properly defined fields/columns for id (primary key), name, quest, color, and comment.

Create a form function called KnightForm that prompts for and accepts the parameters of a Knight (name, quest, color, and comment) and returns a Knight instance.

Create convenience functions to drop the table, create the table, and to quit the program (closing the SQLAlchemy session).

Create a menu system that allows you to list all knights, add a new knight, and quit the program. Use the form to produce a knight for the add function

Create a main function that uses a while loop to display the menu.

If time permits, add a menu item to your app that searches the knight database by name.