FREE eBook

LEARNING sqlalchemy

Free unaffiliated eBook created from
Stack Overflow contributors.

#sqlalchem

y

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sqlalchemy

It is an unofficial and free sqlalchemy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sqlalchemy.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with sqlalchemy

Remarks

SQLALCHEMY'S PHILOSOPHY

From the SQLAlchemy Website:

> SQL databases behave less like object collections the more size and performance start to matter; object collections behave less like tables and rows the more abstraction starts to matter. SQLAlchemy aims to accommodate both of these principles.
>
> SQLAlchemy considers the database to be a relational algebra engine, not just a collection of tables. Rows can be selected from not only tables but also joins and other select statements; any of these units can be composed into a larger structure. SQLAlchemy's expression language builds on this concept from its core.
>
> SQLAlchemy is most famous for its object-relational mapper (ORM), an optional component that provides the data mapper pattern, where classes can be mapped to the database in open ended, multiple ways - allowing the object model and database schema to develop in a cleanly decoupled way from the beginning.
>
> SQLAlchemy's overall approach to these problems is entirely different from that of most other SQL / ORM tools, rooted in a so-called complimentarity- oriented approach; instead of hiding away SQL and object relational details behind a wall of automation, all processes are fully exposed within a series of composable, transparent tools. The library takes on the job of automating redundant tasks while the developer remains in control of how the database is organized and how SQL is constructed.
>
> The main goal of SQLAlchemy is to change the way you think about databases and SQL!

Versions

Version	Release Status	Change Log	Release Date
1.1	Beta	1.1	2016-07-26
1.0	Current Release	1.0	2015-04-16
0.9	Maintenance	0.9	2013-12-30
0.8	Security	0.8	2013-03-09

Examples

Installation or Setup

```
pip install sqlalchemy
```

For most common applications, particularly web applications, it is usually recommended that beginners consider using a supplementary library, such as `flask-sqlalchemy`.

```
pip install flask-sqlalchemy
```

Hello, World! (SQLAlchemy Core)

This example shows how to create a table, insert data, and select from the database using SQLAlchemy Core. For information re: the SQLAlchemy ORM, see here.

First, we'll need to connect to our database.

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://')
```

> The engine is the starting point for any SQLAlchemy application. It's a "home base" for the actual database and its DBAPI, delivered to an SQLAlchemy application through a connection pool and a dialect, which describes how to talk to a specific kind of database/DBAPI combination. The Engine references both a dialect and a connection pool, which together interpret the DBAPI's module functions as well as the behaviour of the database.

After creating our engine, we need to define and create our tables.

```
from sqlalchemy import Column, Integer, Text, MetaData, Table

metadata = MetaData()
messages = Table(
    'messages', metadata,
    Column('id', Integer, primary_key=True),
    Column('message', Text),
)

messages.create(bind=engine)
```

To futher explain the MetaData object, see the below from the docs:

> A collection of Table objects and their associated child objects is referred to as database metadata
>
> We define our tables all within a catalog called MetaData, using the Table construct, which resembles regular SQL CREATE TABLE statements.

Now that we have our tables defined and created, we can start inserting data! Inserting involves two steps. Composing the insert construct, and executing the final query.

```
insert_message = messages.insert().values(message='Hello, World!')
engine.execute(insert_message)
```

Now that we have data, we can use the select function to query our data. Column objects are available as named attributes of the c attribute on the Table object, making it easy to select columns directly. Executing this select statement returns a `ResultProxy` object which has access to a few methods, fetchone(), fetchall(), and fetchmany(), all of which return a number of database rows queried in our select statement.

```
from sqlalchemy import select
stmt = select([messages.c.message])
message, = engine.execute(stmt).fetchone()
print(message)
```

```
Hello, World!
```

And that's it! See the SQLAlchemy SQL Expressions Tutorial for more examples and information.

Hello, World! (SQLAlchemy ORM)

This example shows how to create a table, insert data, and select from the database using the **SQLAlchemy ORM**. For information re: SQLAlchemy Core, see here.

First things first, we need to connect to our database, which is identical to how we would connect using SQLAlchemy Core (Core).

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://')
```

After connecting and creating our engine, we need to define and create our tables. This is where the SQLAlchemy ORM language starts to differ greatly from Core. In ORM, the table creation and definition process begins by defining the tables and the classes we'll use to map to those tables. This process is done in one step in ORM, which SQLAlchemy calls the Declarative system.

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

Now that our base mapper is declared, we can subclass from it to build our declarative mappings, or `models`.

```
from sqlalchemy import Column, Integer, String

class Message(Base):
    __tablename__ = 'messages'

    id = Column(Integer, primary_key=True)
    message = Column(String)
```

Using the declarative base class, we end up creating a `Table` and `Mapper` object. From the docs:

> The Table object is a member of a larger collection known as MetaData. When using Declarative, this object is available using the .metadata attribute of our declarative base class.

With that in mind, to create all tables that do not yet exist, we can call the below command, which utilizes SQLAlchemy Core's MetaData registry.

```
Base.metadata.create_all(engine)
```

Now that our tables are mapped and created, we can insert data! Inserting is done through the creation of mapper instances.

```
message = Message(message="Hello World!")
message.message # 'Hello World!
```

At this point, all we have is an instance of message at the level of the ORM abstraction level, but nothing has been saved to the database yet. To do this, first we need to create a session.

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

This session object is our database handler. As per the SQLAlchemy docs:

> it retrieves a connection from a pool of connections maintained by the Engine, and holds onto it until we commit all changes and/or close the session object.

Now that we have our session, we can add our new message to the session and commit our changes to the database.

```
session.add(message)
session.commit()
```

Now that we have data, we can take advantage of the ORM query language to pull up our data.

```
query = session.query(Message)
instance = query.first()
print (instance.message) # Hello World!
```

But thats just the beginning! There are much more features that can be used to compose queries, like `filter`, `order_by`, and much more. See the SQLAlchemy ORM Tutorial for more examples and information.

Read Getting started with sqlalchemy online: https://riptutorial.com/sqlalchemy/topic/1697/getting-started-with-sqlalchemy

Chapter 2: Connecting

Examples

Engine

The engine is used to connect to different databases using a connection URL:

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://user:pass@localhost/test')
```

Note, however, that the engine does not actually establish a connection until it is first used.

The engine automatically creates a connection pool, but opens new connections lazily (i.e. SQLAlchemy won't open 5 connections if you only ask for one).

Using a Connection

You can open a connection (i.e. request one from the pool) using a context manager:

```
with engine.connect() as conn:
    result = conn.execute('SELECT price FROM products')
    for row in result:
        print('Price:', row['price'])
```

Or without, but it must be closed manually:

```
conn = engine.connect()
result = conn.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
conn.close()
```

Implicit Execution

If you only want to execute a single statement, you can use the engine directly and it will open and close the connection for you:

```
result = engine.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
```

Transactions

You can use `engine.begin` to open a connection and begin a transaction that will be rolled back if an exception is raised, or committed otherwise. This is an implicit way of using a transaction, since

you don't have the option of rolling back manually.

```
with engine.begin() as conn:
    conn.execute(products.insert(), price=15)
```

More explicitly, you can begin a transaction using a connection:

```
with conn.begin() as trans:
    conn.execute(products.insert(), price=15)
```

Note that we still call `execute` on the connection. As before, this transaction will be committed or rolled back if an exception is raised, but we also have access to the transaction, allowing us to rollback manually using `trans.rollback()`.

This could be done more explicitly like so:

```
trans = conn.begin()
try:
    conn.execute(products.insert(), price=15)
    trans.commit()
except:
    trans.rollback()
    raise
```

Read Connecting online: https://riptutorial.com/sqlalchemy/topic/2025/connecting

Chapter 3: Flask-SQLAlchemy

Remarks

Flask-SQLAlchemy adds some additional functionality such as automatic destruction of the session assuming some things for you which are very often not what you need.

Examples

A Minimal Application

For the common case of having one Flask application all you have to do is to create your Flask application, load the configuration of choice and then create the SQLAlchemy object by passing it the application.

Once created, that object then contains all the functions and helpers from both sqlalchemy and sqlalchemy.orm. Furthermore it provides a class called Model that is a declarative base which can be used to declare models:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

Read Flask-SQLAlchemy online: https://riptutorial.com/sqlalchemy/topic/4601/flask-sqlalchemy

Chapter 4: SQLAlchemy Core

Examples

Converting result to dict

In SQLAlchemy core, the result is `RowProxy`. In cases where you want an explicit dictionary, you can call `dict(row)`.

First the setup for the example:

```
import datetime as dt
from sqlalchemy import (
    Column, Date, Integer, MetaData, Table, Text, create_engine, select)

metadata = MetaData()
users = Table(
    'users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', Text, nullable=False),
    Column('birthday', Date),
)

engine = create_engine('sqlite://')
metadata.create_all(bind=engine)

engine.execute(users.insert(), name='Alice', birthday=dt.date(1990, 1, 1))
```

Then to create a dictionary from a result row:

```
with engine.connect() as conn:
    result = conn.execute(users.select())
    for row in result:
        print(dict(row))

    result = conn.execute(select([users.c.name, users.c.birthday]))
    for row in result:
        print(dict(row))
```

Read SQLAlchemy Core online: https://riptutorial.com/sqlalchemy/topic/2022/sqlalchemy-core

Chapter 5: The ORM

Remarks

The SQLAlchemy ORM is built on top of SQLAlchemy Core. For example, although model classes use `Column` objects, they are part of the core and more relevant documentation will be found there.

The main parts of the ORM are the session, query, and mapped classes (typically using the declarative extension in modern SQLAlchemy.)

Examples

Converting a query result to dict

First the setup for the example:

```
import datetime as dt
from sqlalchemy import Column, Date, Integer, Text, create_engine, inspect
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
Session = sessionmaker()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

engine = create_engine('sqlite://')
Base.metadata.create_all(bind=engine)
Session.configure(bind=engine)

session = Session()
session.add(User(name='Alice', birthday=dt.date(1990, 1, 1)))
session.commit()
```

If you're querying columns individually, the row is a `KeyedTuple` which has an `_asdict` method. The method name starts with a single underscore, to match the `namedtuple` API (it's not private!).

```
query = session.query(User.name, User.birthday)
for row in query:
    print(row._asdict())
```

When using the ORM to retrieve objects, this is not available by default. The SQLAlchemy inspection system should be used.

```
def object_as_dict(obj):
```

```
    return {c.key: getattr(obj, c.key)
            for c in inspect(obj).mapper.column_attrs}

query = session.query(User)
for user in query:
    print(object_as_dict(user))
```

Here, we created a function to do the conversion, but one option would be to add a method to the base class.

Instead of using `declarative_base` as above, you can create it from your own class:

```
from sqlalchemy.ext.declarative import as_declarative

@as_declarative()
class Base:
    def _asdict(self):
        return {c.key: getattr(self, c.key)
                for c in inspect(self).mapper.column_attrs}
```

Filtering

Given the following model

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)
```

You can filter columns in the query:

```
import datetime as dt
session.query(User).filter(User.name == 'Bob')
session.query(User).filter(User.birthday < dt.date(2000, 1, 1))
```

For the first case, there is a shortcut:

```
session.query(User).filter_by(name='Bob')
```

Filters can be composed using an AND relation by chaining the `filter` method:

```
(session.query(User).filter(User.name.like('B%'))
                    .filter(User.birthday < dt.date(2000, 1, 1)))
```

Or more flexibly, using the overloaded bitwise operators `&` and `|`:

```
session.query(User).filter((User.name == 'Bob') | (User.name == 'George'))
```

Don't forget the inner parentheses to deal with operator precedence.

Order By

Given a basic model:

```
class SpreadsheetCells(Base):
    __tablename__ = 'spreadsheet_cells'

    id = Column(Integer, primary_key=True)
    y_index = Column(Integer)
    x_index = Column(Integer)
```

You can retrieve an ordered list by chaining the `order_by` method.

```
query = session.query(SpreadsheetCells).order_by(SpreadsheetCells.y_index)
```

This could be chained on after a `filter`,

```
query = session.query(...).filter(...).order_by(...)
```

or to further compose an existing query.

```
query = session.query(...).filter(...)
ordered_query = query.order_by(...)
```

You can also determine the sort direction in one of two ways:

1. Accessing the field properties `asc` and `dsc`:

```
query.order_by(SpreadsheetCells.y_index.desc()) # desc
query.order_by(SpreadsheetCells.y_index.asc()) # asc
```

2. Using the asc and desc module functions:

```
from sqlalchemy import asc, desc

query.order_by(desc(SpreadsheetCells.y_index)) # desc
query.order_by(asc(SpreadsheetCells.y_index)) # asc
```

Accessing query results

Once you have a query, you can do more with it than just iterating the results in a for loop.

Setup:

```
from datetime import date

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
```

```
    name = Column(Text, nullable=False)
    birthday = Column(Date)

# Find users who are older than a cutoff.
query = session.query(User).filter(User.birthday < date(1995, 3, 3))
```

To return the results as a list, use `all()`:

```
reslist = query.all() # all results loaded in memory
nrows = len(reslist)
```

You can get a count using `count()`:

```
nrows = query.count()
```

To get only the first result, use `first()`. This is most useful in combination with `order_by()`.

```
oldest_user = query.order_by(User.birthday).first()
```

For queries that should return only one row, use `one()`:

```
bob = session.query(User).filter(User.name == 'Bob').one()
```

This raises an exception if the query returns multiple rows or if it returns none. If the row might not exist yet, use `one_or_none()`:

```
bob = session.query(User).filter(User.name == 'Bob').one_or_none()
if bob is None:
    create_bob()
```

This will still raise an exception if multiple rows have the name 'Bob'.

Read The ORM online: https://riptutorial.com/sqlalchemy/topic/2020/the-orm

Chapter 6: The Session

Remarks

A session keeps track of ORM objects and their changes, manages transactions and is used to perform queries.

Examples

Creating a Session

A session is usually obtained using `sessionmaker`, which creates a `Session` class unique to your application. Most commonly, the `Session` class is bound to an engine, allowing instances to use the engine implicitly.

```
from sqlalchemy.orm import sessionmaker

# Initial configuration arguments
Session = sessionmaker(bind=engine)
```

The `engine` and `Session` should only be created once.

A session is an instance of the class we created:

```
# This session is bound to provided engine
session = Session()
```

`Session.configure()` can be used to configure the class later, e.g. application startup rather than import time.

```
Session = sessionmaker()

# later
Session.configure(bind=engine)
```

Arguments passed to `Session` directly override the arguments passed to `sessionmaker`.

```
session_bound_to_engine2 = Session(bind=engine2)
```

Adding Instances

New or detached objects may be added to the session using `add()`:

```
session.add(obj)
```

A sequence of objects may be added using `add_all()`:

```
session.add_all([obj1, obj2, obj3])
```

An INSERT will be emitted to the database during the next flush, which happens automatically. Changes are persisted when the session is committed.

Deleting Instances

To delete persisted objects use `delete()`:

```
session.delete(obj)
```

Actual deletion from the database will happen on next flush.

Read The Session online: https://riptutorial.com/sqlalchemy/topic/2258/the-session

Credits

S. No	Chapters	Contributors
1	Getting started with sqlalchemy	adarsh, Community, Mattew Whitt, RazerM, Stephen Fuhry
2	Connecting	RazerM
3	Flask-SQLAlchemy	Ilya Rusin
4	SQLAlchemy Core	RazerM
5	The ORM	Ilja Everilä, Mattew Whitt, RazerM, Tom Hunt
6	The Session	Ilja Everilä, RazerM