Understanding Python SQLAlchemy's Session

This article is part 8 of 11 in the series Python SQLAlchemy Tutorial

What are SQLAlchemy Sessions? What does the Session do?

One of the core concepts in SQLAlchemy is the `Session`. A `Session` establishes and maintains all conversations between your program and the databases. It represents an intermediary zone for all the Python model objects you have loaded in it. It is one of the entry points to initiate a query against the database, whose results are populated and mapped into unique objects within the `Session`. A unique object is the only object in the `Session` with a particular primary key.

A typical lifespan of a `Session` looks like this:

- A `Session` is constructed, at which point it is not associated with any model objects.
- The `Session` receives query requests, whose results are persisted / associated with the `Session`.
- Arbitrary number of model objects are constructed and then added to the `Session`, after which point the `Session` starts to maintain and manage those objects.
- Once all the changes are made against the objects in the `Session`, we may decide to `commit` the changes from the `Session` to the database or `rollback` those changes in the `Session`. `Session.commit()` means that the changes made to the objects in the `Session` so far will be persisted into the database while `Session.rollback()` means those changes will be discarded.
- `Session.close()` will close the `Session` and its corresponding connections, which means we are done with the `Session` and want to release the connection object associated with it.

Understanding SQLAlchemy Sessions by Examples

Let's use a simple example to illustrate how to use `Session` to insert objects into the databases.

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

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)


from sqlalchemy import create_engine
engine = create_engine('sqlite:///')

from sqlalchemy.orm import sessionmaker

# Construct a sessionmaker object
session = sessionmaker()

# Bind the sessionmaker to engine
session.configure(bind=engine)

# Create all the tables in the database which are
# defined by Base's subclasses such as User
Base.metadata.create_all(engine)
```

Create and Persist Session Objects

Once we have a `session`, we can create objects and add them to the `session`.

```
# Make a new Session object
s = session()
john = User(name='John')

# Add User john to the Session object
s.add(john)

# Commit the new User John to the database
s.commit()
```

Let's insert another user Mary and inspect the new object's `id` at each step during the insertion process.

```
>>> mary = User(name='Mary')
>>> print(mary.id, mary.name)
(None, 'Mary')
>>> s.add(mary)
>>> print(mary.id, mary.name)
(None, 'Mary')
>>> s.commit()
>>> print(mary.id, mary.name)
(1, u'Mary')
```

Notice that `mary.id` is `None` before `s.commit()` is called. Why? Because the object `mary` has not been committed to the database when it's constructed and added to `s`, it does not have a primary key assigned by the underlying SQLite database. Once the new object `mary` is committed by `s`, then it is assigned a `id` value by the underlying SQLite database.

Query Objects

Once we have both John and Mary in the database, we can query them using a `Session`.

```
>>> mary = s.query(User).filter(User.name == 'Mary').one()
>>> john = s.query(User).filter(User.name == 'John').one()
>>> mary.id
2
>>> john.id
1
```

As you can see, the queried objects have valid `id` values from the database.

Update Objects

We can change the name of `Mary` just like changing the attribute of a normal Python object, as long as we remember to call `session.commit()` at the end.

```
>>> mary.name = 'Mariana'
>>> s.commit()
>>> mary.name
u'Mariana'
>>> s.query(User).filter(User.name == 'Mariana').one()
>>>
>>> mary.name = 'Mary'
>>> s.commit()
>>> s.query(User).filter(User.name == 'Mariana').one()
Traceback (most recent call last):
......
sqlalchemy.orm.exc.NoResultFound: No row was found for one()
>>> s.query(User).filter(User.name == 'Mary').one()
```

Delete Objects

Now we have two `User` objects persisted in the database, `Mary` and `John`. We are going to delete them by calling `delete()` of the session object.

```
>>> s.delete(mary)
>>> mary.id
2
>>> s.commit()
>>> mary

>>> mary.id
2
>>> mary._sa_instance_state.persistent
False  # Mary is not persistent in the database anymore since she has been deleted by the session
```

Since `Mary` has been marked for deletion by the session and the deletion has been committed by the session into the database, we won't be able to find `Mary` in the database anymore.

```
>>> mary = s.query(User).filter(User.name == 'Mary').one()
Traceback (most recent call last):
......
    raise orm_exc.NoResultFound("No row was found for one()")
sqlalchemy.orm.exc.NoResultFound: No row was found for one()
```

Session Object States

Since we have already seen an `Session` object in action, it's important to also know the four different states of session objects:

- *Transient*: an instance that's not included in a session and has not been persisted to the database.
- *Pending*: an instance that has been added to a session but not persisted to a database yet. It will be persisted to the database in the next `session.commit()`.
- *Persistent*: an instance that has been persisted to the database and also included in a session. You can make a model object persistent by committing it to the database or query it from the database.
- *Detached*: an instance that has been persisted to the database but not included in any sessions.

Let's use `sqlalchemy.inspect` to take a look at the states of a new `User` object `david`.

```
>>> from sqlalchemy import inspect
>>> david = User(name='David')
>>> ins = inspect(david)
>>> print('Transient: {0}; Pending: {1}; Persistent: {2}; Detached: {3}'.format(ins.transient, ins.pending, ins.pe
rsistent, ins.detached))
Transient: True; Pending: False; Persistent: False; Detached: False
>>> s.add(david)
>>> print('Transient: {0}; Pending: {1}; Persistent: {2}; Detached: {3}'.format(ins.transient, ins.pending, ins.pe
rsistent, ins.detached))
Transient: False; Pending: True; Persistent: False; Detached: False
>>> s.commit()
>>> print('Transient: {0}; Pending: {1}; Persistent: {2}; Detached: {3}'.format(ins.transient, ins.pending, ins.pe
rsistent, ins.detached))
Transient: False; Pending: False; Persistent: True; Detached: False
>>> s.close()
>>> print('Transient: {0}; Pending: {1}; Persistent: {2}; Detached: {3}'.format(ins.transient, ins.pending, ins.pe
rsistent, ins.detached))
Transient: False; Pending: False; Persistent: False; Detached: True
```

Notice the change of `david`'s state progressing from *Transient* to *Detached* at each step of the insertion process. It's important to become familiar with these states of the objects because a slight misunderstanding may lead to hard-to-find bugs in a program.

Scoped Session vs. Normal Session

So far, the session object we constructed from the `sessionmaker()` call and used to communicate with our database is a normal session. If you call `sessionmaker()` a second time, you will get a new session object whose states are independent of the previous session. For example, suppose we have two session objects constructed in the following way:

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

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)


from sqlalchemy import create_engine
engine = create_engine('sqlite:///')

from sqlalchemy.orm import sessionmaker
session = sessionmaker()
session.configure(bind=engine)
Base.metadata.create_all(engine)

# Construct the first session object
s1 = session()
# Construct the second session object
s2 = session()
```

Then, we won't be able to add the same `User` object to both `s1` and `s2` at the same time. In other words, an object can only be attached at most one unique `session` object.

```
>>> jessica = User(name='Jessica')
>>> s1.add(jessica)
>>> s2.add(jessica)
Traceback (most recent call last):
......
sqlalchemy.exc.InvalidRequestError: Object '' is already attached to session '2' (this is '3')
```

If the session objects are retrieved from a `scoped_session` object, however, then we don't have such an issue since the `scoped_session` object maintains a registry for the same session object.

```
>>> session_factory = sessionmaker(bind=engine)
>>> session = scoped_session(session_factory)
>>> s1 = session()
>>> s2 = session()
>>> jessica = User(name='Jessica')
>>> s1.add(jessica)
>>> s2.add(jessica)
>>> s1 is s2
True
>>> s1.commit()
>>> s2.query(User).filter(User.name == 'Jessica').one()
```

Notice that `s1` and `s2` are the same session object since they are both retrieved from a `scoped_session` object who maintains a reference to the same session object.

Summary and Tips

In this article, we reviewed how to use `SQLAlchemy`'s `Session` and the four different states of a model object. Since *unit-of-work* is a core concept in SQLAlchemy, it's crucial to fully understand and be familiar with how to use `Session` and the four different states of a model object. In the next article, we will show you how to utilize the `Session` to manage complex model objects and avoid common bugs.