Sheena
Follow
Sign in
Published Mar 18, 2015Last updated Feb 10, 2017

A SQLAlchemy Cheat Sheet

Introduction

SQLAlchemy is a deep and powerful thing made up of many layers. This cheat sheet sticks to parts of the ORM (Object Relational Mapper) layer,and aims to be a reference not a tutorial. That said, if you are familiar with SQL then this cheat sheet should get you well on your way to understanding SQLAlchemy.

Basic Models

One model is used to describe one database table. For example:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session,sessionmaker
from zope.sqlalchemy import ZopeTransactionExtension
from sqlalchemy import (
    Column,
    Integer,
    String,
    Boolean,
    ForeignKey,
    DateTime,
    Sequence,
    Float
)
import datetime

DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
Base = declarative_base()

class Book(Base):  #<-------------------------
    __tablename__  = "books"    #matches the name of the actual database table
    id             = Column(Integer,Sequence('book_seq'),primary_key=True) # plays nice with all major database engines
    name           = Column(String(50))                                    # string column need lengths
    author_id      = Column(Integer,ForeignKey('authors.id'))              # assumes there is a table in the database called 'authors' that has an 'id' column
    price          = Column(Float)
    date_added     = Column(DateTime, default=datetime.datetime.now)       # defaults can be specified as functions
    promote        = Column(Boolean,default=False)                         #     or as values
```

Queries and Interactions

Selecting and Filtering

```
#fetch everything
lBooks = DBSession.query(Book)  #returns a Query object.
for oBook in lBooks:
    print oBook.name

#simple filters
lBooks = DBSession.query(Book).filter_by(author_id=1) #returns all the books for a specific author

#more complex filters
lBooks = DBSession.query(Book).filter(Book.price<20) #returns all the books with price <20. Note we use filter, not filter_by

#filters can be combined
lBooks = DBSession.query(Book).filter_by(author_id=1).filter(Book.price<20) #all books by a specific author, with price<20

#logical operations can be used in filters
from sqlalchemy import or_
lBooks = DBSession.query(Book).filter(or_(Book.price<20,promote==True)) # returns all books  that cost less than 20 OR are being promoted

#ordering
from sqlalchemy import desc
DBSession.query(Book).order_by(Book.price) #get all books ordered by price
DBSession.query(Book).order_by(desc(Book.price)) #get all books ordered by price descending

#other useful things
DBSession.query(Book).count() #returns the number of books
DBSession.query(Book).offset(5) #offset the result by 5
DBSession.query(Book).limit(5) # return at most 5 books
DBSession.query(Book).first() #return the first book only or None
DBSession.query(Book).get(8) #return the Book with primary key = 8, or None
```

Relationships

Relationships between SQL tables are described in terms of foreign key relationships. From the example, the `books` table has a foreign key field pointing to the id field of the authors table. SQLAlchemy makes leveraging and examining those relationships pretty straight forward.

One to many relationships

Assume we are keeping track of the books of various authors. *One* author can have *many* books.

```
class Book(Base):
    __tablename__  = "books"    #matches the name of the actual database table
    id             = Column(Integer,Sequence('book_seq'),primary_key=True)
    name           = Column(String(50))
    author_id      = Column(Integer,ForeignKey('authors.id'))
    author = relationship("Author",backref="books")            # <-----------------------

class Author(Base):
    __tablename__  = "books"    #matches the name of the actual database table
    id             = Column(Integer,Sequence('book_seq'),primary_key=True)
    name           = Column(String(50))
```

The marked line configures the relationship between the models. Note that `"Author"` is a string. It doesn't need to be, it can also be a class. Using a string here removes the possibility of certain NameErrors. Note that the relationship is configured in both directions in one line. A book's author is accessable via the `author` attribute, and an author's books are accessable via the author's `books` attribute.

Here are a few ways you can make use of the relationship once it is configured:

```
oBook = DBSession.query(Book).filter_by(name="Harry Potter and the methods of rationality").first()
oAuthor = oBook.author   # oAuthor is now an Author instance. oAuthor.id == oBook.author_id

#it works the other way as well
oAuthor = DBSession.query(Author).filter_by(name="Orsan Scott Card")
for oBook in oAuthor.books:
    print oBook.name

#adding a new book
oNewBook = Book()
oBook.name = "Ender's Game"
oBook.author = oAuthor

#adding a new book in a different way...
oNewBook = Book()
oBook.name = "Ender's Shadow"
oAuthor.books.append(oBook)
```

One to one relationships

```
class Parent(Base):
    __tablename__ = 'parent'
    id = ColumnColumn(Integer,Sequence('p_seq'),primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref=backref("parent", uselist=False)) # <------

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer,Sequence('c_seq'),primary_key=True)
```

Note that the line that configures the relationship has an expression instead of just a string for the backref argument. If the string `"parent"` was used then it would be a normal many to one relationship. We can make use of this like so:

```
oChild = DBSession.query(Child).get(1)
oParent = oChild.parent

oParent2 = Parent()
oParent.child = Child()
```

Many to many relationships

A many to many relationship requires an extra table to create mappings between lines. There are two ways of doing this:

First, just using models:

```
class Category(Base):
    __tablename__ = 'categories'
    id = Column(Integer,Sequence('cat_seq'),primary_key=True)
    name = Column(String(20))

class Product(Base):
    __tablename__ = 'products'
    id = Column(Integer,Sequence('prod_seq'),primary_key=True)
    name = Column(String(20))

class Map(Base):
    __tablename__ = 'map'
    id = Column(Integer,Sequence('map_seq'),primary_key=True)
    cat_id = Column(Integer,ForeignKey('categories.id'))
    prod_id = Column(Integer,ForeignKey('products.id'))
```

Here you can specify relationships on the `Map` class. The benefit of this approach is that you can instantiate the `Map` class, this is useful if you want to interact with `Map` objects in any non-trivial way.

This next approach is better if your map table is only a map table and requires no complex interactions:

```
map_table = Table('maps', Base.metadata,
    Column('cat_id', Integer, ForeignKey('categories.id')),
    Column('prod_id', Integer, ForeignKey('products.id'))
)

class Category(Base):
    __tablename__ = 'categories'
    id = Column(Integer,Sequence('cat_seq'),primary_key=True)
    name = Column(String(20))

    products = relationship("Product",
                    secondary=map_table,   # you can also use the string name of the table, "maps", as the secondary
                    backref="categories")

class Product(Base):
    __tablename__ = 'products'
    id = Column(Integer,Sequence('prod_seq'),primary_key=True)
    name = Column(String(20))
```

You can make use of the relationship like this:

```
#construct a category and add some products to it
oCat = Category()
oCat.name = "Books"

oProduct = Product()
oProduct.name = "Ender's Game - Orsan Scott Card"
oCat.products.append(oProduct)

oProduct = Product()
oProduct.name = "Harry Potter and the methods of Rationality"
oProduct.categories.append(oCat)

# interact with products from an existing category
for oProduct in oCat.products:
    print oProduct.name

#interact with categories of an existing product
oProduct = DBSession.query(Product).filter_by(name="")
for oCat in oProduct.categories:
    print oCat.name
```

Self referential relationships

Sometimes you have a table with a foreign key pointing at the same table. For example, say we have a bunch of nodes in a directed tree. A node can have many child nodes but at most one parent

```
class TreeNode(Base):
    __tablename__ = 'nodes'
    id = Column(Integer,Sequence('node_seq'),primary_key=True)
    parent_id = Column(Integer,ForeignKey('nodes.id'))
    name = Column(String(20))

    children = relationship("TreeNode",
                backref=backref('parent', remote_side=[id])
            )
```

You can make use of this relationship like any many to one relationship:

```
#fetch the root node (assume there is one node with no parents)
oRootNode = DBSession.query(TreeNode).filter_by(parent_id=None).first()

#interact with children of existing node
for oChild in oRootNode.children:
    print oChild.name

#create new relationships

oParent = TreeNode()
oParent.name = "parent"
oRootNode.children.append(oParent)

oChild = TreeNode()
oChild.name = "Child"
oChild.parent = oParent
```

Multiple relationships with the same table

```
class WikiPost(Base):
    __tablename__ = 'posts'
    id = Column(Integer,Sequence('post_seq'),primary_key=True)
    name = Column(String(20))
    author_id = Column(Integer,ForeignKey('users.id'))
    editor_id = Column(Integer,ForeignKey('users.id'))

    editor = relationship("User", primaryjoin = "WikiPost.editor_id == User.id",backref="edited_posts")
    author = relationship("User", primaryjoin = "WikiPost.author_id == User.id",backref="authored_posts")

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer,Sequence('usr_seq'),primary_key=True)
    name = Column(String(20))
```

You can interact with this just like two many to one relationships.

```
oAuthor = DBSession.query(User).filter_by(name="Sheena O'Connell")

#an author writes a post
oPost = WikiPost()
oPost.name = "Sqlalchemy Cheat Sheet"
oPost.author = oAuthor

#later on another user edits it
oEditor = DBSession.query(User).filter_by(name="Yi-Jirr Chen")
oEditor.edited_posts.append(oPost)

#interact with existing relationships
for oPost in oAuthor.authored_posts:
    print oPost.name

for oPost in oEditor.edited_posts:
    print oPost.name
```

Engine Configuration

Connection Strings

```
#the general form of a connection string:
`dialect+driver://username:password@host:port/database`

#SQLITE:
'sqlite:///:memory:' #store everything in memory, data is lost when program exits
'sqlite:////absolute/path/to/project.db')  #Unix/Mac
'sqlite:///C:\\path\\to\\project.db' #Windows
r'sqlite:///C:\path\to\project.db' #Windows alternative

#PostgreSQL
```

```
'postgresql://user:pass@localhost/mydatabase'
'postgresql+psycopg2://user:pass@localhost/mydatabase'
'postgresql+pg8000://user:pass@localhost/mydatabase'

#Oracle
'oracle://user:pass@127.0.0.1:1521/sidname'
'oracle+cx_oracle://user:pass@tnsname'

#Microsoft SQL Server
'mssql+pyodbc://user:pass@mydsn'
'mssql+pymssql://user:pass@hostname:port/dbname'
```

Engine, Session and Base

```
#set up the engine
engine = create_engine(sConnectionString, echo=True)   #echo=True makes the sql commands issued by sqlalchemy get output to the console, useful for debugging

#bind the dbsession to the engine
DBSession.configure(bind=engine)

#now you can interact with the database if it exists

#import all your models then execute this to create any tables that don't yet exist. This does not handle migrations
Base.metadata.create_all(engine)
```

Conclusion

There are many more ways in which you can configure relationships, many more ways to query a database, and many topics that this cheat sheet just didn't cover. SQLAlchemy is pretty huge, but for many applications you don't need to do anything more complex than what is shown here. If you came here looking to learn how to use SQLAlchemy and you feel you are in a position where you can make use of the methods I described here then I would suggest as a next step you read up a little bit on how to use the session to manage transactions.

PythonSQLsqlalchemy

Star or share to let the writer know you enjoyed this post!

Sheena
I'm a software engineer. I do most of my work in Python2.7 and I'm currently involved in creating tools that make hadoop more useful. I have acted as a mentor a few times to junior employees and I'm told I'm a good teacher and I d...
Follow
Jose A Dianes
Spark & Python: SQL & DataFrames
Jose A Dianes
Data Science with Python & R: Dimensionality Reduction and Clustering
Sumit Raj
10 Neat Python Tricks Beginners Should Know