Object Relational Tutorial

The SQLAlchemy Object Relational Mapper presents a method of associating user-defined Python classes with database tables, and instances of those classes (objects) with rows in their corresponding tables. It includes a system that transparently synchronizes all changes in state between objects and their related rows, called a unit of work, as well as a system for expressing database queries in terms of the user defined classes and their defined relationships between each other.

The ORM is in contrast to the SQLAlchemy Expression Language, upon which the ORM is constructed. Whereas the SQL Expression Language, introduced in SQL Expression Language Tutorial, presents a system of representing the primitive constructs of the relational database directly without opinion, the ORM presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language.

While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. One approaches the structure and content of data from the perspective of a user-defined domain model which is transparently persisted and refreshed from its underlying storage model. The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.

A successful application may be constructed using the Object Relational Mapper exclusively. In advanced situations, an application constructed with the ORM may make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

The following tutorial is in doctest format, meaning each `>` line represents something you can type at a Python command prompt, and the following text represents the expected return value.

Version Check

A quick check to verify that we are on at least **version 1.1** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
1.1.0
```

Connecting

For this tutorial we will use an in-memory-only SQLite database. To connect we use `c`:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The `e` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `l` module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to `F`. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

The return value of `c` is an instance of `E`, and it represents the core interface to the database, adapted through a dialect that handles the details of the database and DBAPI in use. In this case the SQLite dialect will interpret instructions to the Python built-in `s` module.

The first time a method like `E` or `E` is called, the `E` establishes a real DBAPI connection to the database, which is then used to emit the SQL. When using the ORM, we typically don't use the `E` directly once created; instead, it's used behind the scenes by the ORM as we'll see shortly.

Lazy Connecting

The `E`, when first returned by `c`, has not actually tried to connect to the database yet; that happens only the first time it is asked to perform a task against the database.

See also

Database Urls - includes examples of `c` connecting to several kinds of databases with links to more information.

Declare a Mapping

When using the ORM, the configurational process starts by describing the database tables we'll be dealing with, and then by defining our own classes which will be mapped to those tables. In modern SQLAlchemy, these two tasks are usually performed together, using a system known as Declarative, which allows us to create classes that include directives to describe the actual database table they will be mapped to.

Classes mapped using the Declarative system are defined in terms of a base class which maintains a catalog of classes and tables relative to that base - this is known as the **declarative base class**. Our application will usually have just one instance of this base in a commonly imported module. We create the base class using the `d` function, as follows:

```
>>> from sqlalchemy.ext.declarative import declarative_base

>>> Base = declarative_base()
```

Now that we have a "base", we can define any number of mapped classes in terms of it. We will start with just a single table called `u`, which will store records for the end-users using our application. A new class called `U` will be the class to which we map this table. Within the class, we define details about the table to which we'll be mapping, primarily the table name, and names and datatypes of columns:

```
>>> from sqlalchemy import Column, Integer, String
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
...     def __repr__(self):
...        return "<User(name='%s', fullname='%s', password='%s')>" % (
...                             self.name, self.fullname, self.password)
```

A class using Declarative at a minimum needs a `_` attribute, and at least one `C` which is part of a primary key [1]. SQLAlchemy never makes any assumptions by itself about the table to which a class refers, including that it has no built-in conventions for names, datatypes, or constraints. But this doesn't mean boilerplate is required; instead, you're encouraged to create your own automated conventions using helper functions and mixin classes, which is described in detail at Mixin and Custom Base Classes.

Tip

The `U` class defines a `_` method, but note that is **optional**; we only implement it in this tutorial so that our examples show nicely formatted `U` objects.

When our class is constructed, Declarative replaces all the `C` objects with special Python accessors known as descriptors; this is a process known as instrumentation. The "instrumented" mapped class will provide us with the means to refer to our table in a SQL context as well as to persist and load the values of columns from the database.

Outside of what the mapping process does to our class, the class remains otherwise mostly a normal Python class, to which we can define any number of ordinary attributes and methods needed by our application.

[1] For information on why a primary key is required, see How do I map a table that has no primary key?.

Create a Schema

With our `U` class constructed via the Declarative system, we have defined information about our table, known as table metadata. The object used by SQLAlchemy to represent this information for a specific table is called the `T` object, and here Declarative has made one for us. We can see this object by inspecting the `_` attribute:

```
>>> User.__table__
Table('users', MetaData(bind=None),
            Column('id', Integer(), table=<users>, primary_key=True, nullable=False),
            Column('name', String(), table=<users>),
            Column('fullname', String(), table=<users>),
            Column('password', String(), table=<users>), schema=None)
```

When we declared our class, Declarative used a Python metaclass in order to perform additional activities once the class declaration was complete; within this phase, it then created a `T` object according to our specifications, and associated it with the class by constructing a `M` object. This object is a behind-the-scenes object we normally don't need to deal with directly (though it can provide plenty of information about our mapping when we need it).

Classical Mappings

The Declarative system, though highly recommended, is not required in order to use SQLAlchemy's ORM. Outside of Declarative, any plain Python class can be mapped to any `T` using the `m` function directly; this less common usage is described at Classical Mappings.

The `T` object is a member of a larger collection known as `M`. When using Declarative, this object is available using the `.` attribute of our declarative base class.

The `M` is a registry which includes the ability to emit a limited set of schema generation commands to the database. As our SQLite database does not actually have a `u` table present, we can use `M` to issue CREATE TABLE statements to the database for all tables that don't yet exist. Below, we call the `M` method, passing in our `E` as a source of database connectivity. We will see that special commands are first emitted to check for the presence of the `u` table, and following that the actual `C T` statement:

```
>>> Base.metadata.create_all(engine)
SELECT ...
PRAGMA table_info("users")
()
CREATE TABLE users (
    id INTEGER NOT NULL, name VARCHAR,
    fullname VARCHAR,
    password VARCHAR,
    PRIMARY KEY (id)
)
()
COMMIT
```

Minimal Table Descriptions vs. Full Descriptions

Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite and Postgresql, this is a valid datatype, but on others, it's not allowed. So if running this tutorial on one of those databases, and you wish to use SQLAlchemy to issue CREATE TABLE, a "length" may be provided to the `S` type as below:

```
Column(String(50))
```

The length field on `S`, as well as similar precision/scale fields available on `I`, `N`, etc. are not referenced by SQLAlchemy other than when creating tables.

Additionally, Firebird and Oracle require sequences to generate new primary key identifiers, and SQLAlchemy doesn't generate or assume these without being instructed. For that, you use the `S` construct:

```
from sqlalchemy import Sequence
Column(Integer, Sequence('user_id_seq'), primary_key=True)
```

A full, foolproof `T` generated via our declarative mapping is therefore:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, Sequence('user_id_seq'), primary_key=True)
    name = Column(String(50))
    fullname = Column(String(50))
    password = Column(String(12))
```

```
    def __repr__(self):
        return "<User(name='%s', fullname='%s', password='%s')>" % (
                                self.name, self.fullname, self.password)
```

We include this more verbose table definition separately to highlight the difference between a minimal construct geared primarily towards in-Python usage only, versus one that will be used to emit CREATE TABLE statements on a particular set of backends with more stringent requirements.

Create an Instance of the Mapped Class

With mappings complete, let's now create and inspect a `U` object:

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> ed_user.name
'ed'
>>> ed_user.password
'edspassword'
>>> str(ed_user.id)
'None'
```

Even though we didn't specify it in the constructor, the `i` attribute still produces a value of `N` when we access it (as opposed to Python's usual behavior of raising `A` for an undefined attribute). SQLAlchemy's instrumentation normally produces this default value for column-mapped attributes when first accessed. For those attributes where we've actually assigned a value, the instrumentation system is tracking those assignments for use within an eventual INSERT statement to be emitted to the database.

the `_` method

Our `U` class, as defined using the Declarative system, has been provided with a constructor (e.g. `_` method) which automatically accepts keyword names that match the columns we've mapped. We are free to define any explicit `_` method we prefer on our class, which will override the default method provided by Declarative.

Creating a Session

We're now ready to start talking to the database. The ORM's "handle" to the database is the `S`. When we first set up the application, at the same level as our `c` statement, we define a `S` class which will serve as a factory for new `S` objects:

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)
```

In the case where your application does not yet have an `E` when you define your module-level objects, just set it up like this:

```
>>> Session = sessionmaker()
```

Later, when you create your engine with `c`, connect it to the `S` using `c`:

```
>>> Session.configure(bind=engine)  # once engine is available
```

This custom-made `S` class will create new `S` objects which are bound to our database. Other transactional characteristics may be defined when calling `s` as well; these are described in a later chapter. Then, whenever you need to have a conversation with the database, you instantiate a `S`:

```
>>> session = Session()
```

The above `S` is associated with our SQLite-enabled `E`, but it hasn't opened any connections yet. When it's first used, it retrieves a connection from a pool of connections maintained by the `E`, and holds onto it until we commit all changes and/or close

Session Lifecycle Patterns

The question of when to make a `S` depends a lot on what kind of application is being built. Keep in mind, the `S` is just a workspace for your objects, local to a particular database connection - if you think of an application thread as a guest at a dinner party, the `S` is the guest's plate and the objects it holds are the food (and the database...the kitchen?)! More on this topic available at When do I construct a Session, when do I commit it, and when do I close it?.

the session object.

Adding and Updating Objects

To persist our `U` object, we `a` it to our `S`:

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> session.add(ed_user)
```

At this point, we say that the instance is **pending**; no SQL has yet been issued and the object is not yet represented by a row in the database. The `S` will issue the SQL to persist `E J` as soon as is needed, using a process known as a **flush**. If we query the database for `E J`, all pending information will first be flushed, and the query is issued immediately thereafter.

For example, below we create a new `Q` object which loads instances of `U`. We “filter by” the `n` attribute of `e`, and indicate that we’d like only the first result in the full list of rows. A `U` instance is returned which is equivalent to that which we’ve added:

```
>>> our_user = session.query(User).filter_by(name='ed').first() # doctest:+NORMALIZE_WH
>>> our_user
<User(name='ed', fullname='Ed Jones', password='edspassword')>
```

SQL

In fact, the `S` has identified that the row returned is the **same** row as one already represented within its internal map of objects, so we actually got back the identical instance as that which we just added:

```
>>> ed_user is our_user
True
```

The ORM concept at work here is known as an identity map and ensures that all operations upon a particular row within a `S` operate upon the same set of data. Once an object with a particular primary key is present in the `S`, all SQL queries on that `S` will always return the same Python object for that particular primary key; it also will raise an error if an attempt is made to place a second, already-persisted object with the same primary key within the session.

We can add more `U` objects at once using `a`:

```
>>> session.add_all([
...     User(name='wendy', fullname='Wendy Williams', password='foobar'),
...     User(name='mary', fullname='Mary Contrary', password='xxg527'),
...     User(name='fred', fullname='Fred Flinstone', password='blah')])
```

Also, we’ve decided the password for Ed isn’t too secure, so lets change it:

```
>>> ed_user.password = 'f8s7ccs'
```

The `S` is paying attention. It knows, for example, that `E J` has been modified:

```
>>> session.dirty
IdentitySet([<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>])
```

and that three new `U` objects are pending:

```
>>> session.new  # doctest: +SKIP
IdentitySet([<User(name='wendy', fullname='Wendy Williams', password='foobar')>,
<User(name='mary', fullname='Mary Contrary', password='xxg527')>,
<User(name='fred', fullname='Fred Flinstone', password='blah')>])
```

We tell the `S` that we’d like to issue all remaining changes to the database and commit the transaction, which has been in progress throughout. We do this via `c`. The `S` emits the `U` statement for the password change on “ed”, as well as `I` statements for the three new `U` objects we’ve added:

```
>>> session.commit()
```

SQL

`c` flushes whatever remaining changes remain to the database, and commits the transaction. The connection resources referenced by the session are now returned to the connection pool. Subsequent operations with this session will occur in a **new** transaction, which will again re-acquire connection resources when first needed.

If we look at Ed's `i` attribute, which earlier was `N`, it now has a value:

```
>>> ed_user.id # doctest: +NORMALIZE_WHITESPACE
1
```

SQL

After the `S` inserts new rows in the database, all newly generated identifiers and database-generated defaults become available on the instance, either immediately or via load-on-first-access. In this case, the entire row was re-loaded on access because a new transaction was begun after we issued `c`. SQLAlchemy by default refreshes data from a previous transaction the first time it's accessed within a new transaction, so that the most recent state is available. The level of reloading is configurable as is described in *Using the Session*.

Session Object States

As our `U` object moved from being outside the `S`, to inside the `S` without a primary key, to actually being inserted, it moved between three out of four available "object states" - **transient**, **pending**, and **persistent**. Being aware of these states and what they mean is always a good idea - be sure to read Quickie Intro to Object States for a quick overview.

Rolling Back

Since the `S` works within a transaction, we can roll back changes made too. Let's make two changes that we'll revert; `e`'s user name gets set to `E`:

```
>>> ed_user.name = 'Edwardo'
```

and we'll add another erroneous user, `f`:

```
>>> fake_user = User(name='fakeuser', fullname='Invalid', password='12345')
>>> session.add(fake_user)
```

Querying the session, we can see that they're flushed into the current transaction:

```
>>> session.query(User).filter(User.name.in_(['Edwardo', 'fakeuser'])).all()
[<User(name='Edwardo', fullname='Ed Jones', password='f8s7ccs')>, <User(name='fakeuser'
```

SQL

Rolling back, we can see that `e`'s name is back to `e`, and `f` has been kicked out of the session:

```
>>> session.rollback()

>>> ed_user.name
u'ed'
>>> fake_user in session
False
```

SQL

SQL

issuing a SELECT illustrates the changes made to the database:

```
>>> session.query(User).filter(User.name.in_(['ed', 'fakeuser'])).all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

SQL

Querying

A `Q` object is created using the `q` method on `S`. This function takes a variable number of arguments, which can be any combination of classes and class-instrumented descriptors. Below, we indicate a `Q` which loads `U` instances. When evaluated in an iterative context, the list of `U` objects present is returned:

```
>>> for instance in session.query(User).order_by(User.id):
...     print(instance.name, instance.fullname)
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The `Q` also accepts ORM-instrumented descriptors as arguments. Any time multiple class entities or column-based entities are expressed as arguments to the `q` function, the return result is expressed as tuples:

```
>>> for name, fullname in session.query(User.name, User.fullname):
...     print(name, fullname)
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The tuples returned by `Q` are *named* tuples, supplied by the `K` class, and can be treated much like an ordinary Python object. The names are the same as the attribute's name for an attribute, and the class name for a class:

```
>>> for row in session.query(User, User.name).all():
...    print(row.User, row.name)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> ed
<User(name='wendy', fullname='Wendy Williams', password='foobar')> wendy
<User(name='mary', fullname='Mary Contrary', password='xxg527')> mary
<User(name='fred', fullname='Fred Flinstone', password='blah')> fred
```

You can control the names of individual column expressions using the `l` construct, which is available from any `C` -derived object, as well as any class attribute which is mapped to one (such as `U`):

```
>>> for row in session.query(User.name.label('name_label')).all():
...    print(row.name_label)
ed
wendy
mary
fred
```

The name given to a full entity such as `U` , assuming that multiple entities are present in the call to `q` , can be controlled using `a` :

```
>>> from sqlalchemy.orm import aliased
>>> user_alias = aliased(User, name='user_alias')

>>> for row in session.query(user_alias, user_alias.name).all():
...    print(row.user_alias)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

Basic operations with `Q` include issuing LIMIT and OFFSET, most conveniently using Python array slices and typically in conjunction with ORDER BY:

```
>>> for u in session.query(User).order_by(User.id)[1:3]:
...    print(u)
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
```

and filtering results, which is accomplished either with `f` , which uses keyword arguments:

```
>>> for name, in session.query(User.name).\
...             filter_by(fullname='Ed Jones'):
...    print(name)
ed
```

...or `f` , which uses more flexible SQL expression language constructs. These allow you to use regular Python operators with the class-level attributes on your mapped class:

```
>>> for name, in session.query(User.name).\
...             filter(User.fullname=='Ed Jones'):
...    print(name)
ed
```

SQL

The `Q` object is fully **generative**, meaning that most method calls return a new `Q` object upon which further criteria may be added. For example, to query for users named "ed" with a full name of "Ed Jones", you can call `f` twice, which joins criteria using `A` :

```
>>> for user in session.query(User).\
...          filter(User.name=='ed').\
...          filter(User.fullname=='Ed Jones'):
...    print(user)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

SQL

Common Filter Operators

Here's a rundown of some of the most common operators used in `f` :

- `e` :

  ```
  query.filter(User.name == 'ed')
  ```

- `n e` :

  ```
  query.filter(User.name != 'ed')
  ```

- `L` :

  ```
  query.filter(User.name.like('%ed%'))
  ```

- `I` :

  ```
  query.filter(User.name.in_(['ed', 'wendy', 'jack']))

  # works with query objects too:
  query.filter(User.name.in_(
          session.query(User.name).filter(User.name.like('%ed%'))
  ))
  ```

- `N I` :

  ```
  query.filter(~User.name.in_(['ed', 'wendy', 'jack']))
  ```

- `I N` :

  ```
  query.filter(User.name == None)

  # alternatively, if pep8/linters are a concern
  query.filter(User.name.is_(None))
  ```

- `I N N` :

  ```
  query.filter(User.name != None)

  # alternatively, if pep8/linters are a concern
  query.filter(User.name.isnot(None))
  ```

- `A` :

  ```
  # use and_()
  from sqlalchemy import and_
  query.filter(and_(User.name == 'ed', User.fullname == 'Ed Jones'))

  # or send multiple expressions to .filter()
  query.filter(User.name == 'ed', User.fullname == 'Ed Jones')

  # or chain multiple filter()/filter_by() calls
  ```

```
query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')
```

Note

Make sure you use `a` and **not** the Python `a` operator!

- `O` :

```
from sqlalchemy import or_
query.filter(or_(User.name == 'ed', User.name == 'wendy'))
```

Note

Make sure you use `o` and **not** the Python `o` operator!

- `M` :

```
query.filter(User.name.match('wendy'))
```

Note

`m` uses a database-specific `M` or `C` function; its behavior will vary by backend and is not available on some backends such as SQLite.

Returning Lists and Scalars

A number of methods on `Q` immediately issue SQL and return a value containing loaded database results. Here's a brief tour:

- `a` returns a list:

```
>>> query = session.query(User).filter(User.name.like('%ed')).order_by(User.id)
>>> query.all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>,
      <User(name='fred', fullname='Fred Flinstone', password='blah')>]
```

SQL

- `f` applies a limit of one and returns the first result as a scalar:

```
>>> query.first()
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

SQL

- `o` fully fetches all rows, and if not exactly one object identity or composite row is present in the result, raises an error. With multiple rows found:

```
>>> user = query.one()
Traceback (most recent call last):
...
MultipleResultsFound: Multiple rows were found for one()
```

With no rows found:

```
>>> user = query.filter(User.id == 99).one()
Traceback (most recent call last):
...
NoResultFound: No row was found for one()
```

The `o` method is great for systems that expect to handle "no items found" versus "multiple items found" differently; such as a RESTful web service, which may want to raise a "404 not found" when no results are found, but raise an application error when multiple results are found.

- `o` is like `o`, except that if no results are found, it doesn't raise an error; it just returns `N`. Like `o`, however, it does raise an error if multiple results are found.
- `s` invokes the `o` method, and upon success returns the first column of the row:

```
>>> query = session.query(User.id).filter(User.name == 'ed').\
...     order_by(User.id)
>>> query.scalar()
1
```

Using Textual SQL

Literal strings can be used flexibly with `Q`, by specifying their use with the `t` construct, which is accepted by most applicable methods. For example, `f` and `o`:

```
>>> from sqlalchemy import text
>>> for user in session.query(User).\
...             filter(text("id<224")).\
...             order_by(text("id")).all():
...     print(user.name)
ed
wendy
mary
fred
```

Bind parameters can be specified with string-based SQL, using a colon. To specify the values, use the `p` method:

```
>>> session.query(User).filter(text("id<:value and name=:name")).\
...     params(value=224, name='fred').order_by(User.id).one()
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

To use an entirely string-based statement, a `t` construct representing a complete statement can be passed to `f`. Without additional specifiers, the columns in the string SQL are matched to the model columns based on name, such as below where we use just an asterisk to represent loading all columns:

```
>>> session.query(User).from_statement(
...                     text("SELECT * FROM users where name=:name")).\
...                     params(name='ed').all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

Matching columns on name works for simple cases but can become unwieldy when dealing with complex statements that contain duplicate column names or when using anonymized ORM constructs that don’t easily match to specific names. Additionally, there is typing behavior present in our mapped columns that we might find necessary when handling result rows. For these cases, the `t` construct allows us to link its textual SQL to Core or ORM-mapped column expressions positionally; we can achieve this by passing column expressions as positional arguments to the `T` method:

```
>>> stmt = text("SELECT name, id, fullname, password "
...             "FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id, User.fullname, User.password)
>>> session.query(User).from_statement(stmt).params(name='ed').all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

New in version 1.1: The `T` method now accepts column expressions which will be matched positionally to a plain text SQL result set, eliminating the need for column names to match or even be unique in the SQL statement.

When selecting from a `t` construct, the `Q` may still specify what columns and entities are to be returned; instead of `q` we can also ask for the columns individually, as in any other case:

```
>>> stmt = text("SELECT name, id FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id)
>>> session.query(User.id, User.name).\
...          from_statement(stmt).params(name='ed').all()
[(1, u'ed')]
```

See also

Using Textual SQL - The `t` construct explained from the perspective of Core-only queries.

Counting

`Q` includes a convenience method for counting called `c`:

```
>>> session.query(User).filter(User.name.like('%ed')).count()
2
```

SQL

The `c` method is used to determine how many rows the SQL statement would return. Looking at the generated SQL above, SQLAlchemy always places whatever it is we are querying into a subquery, then counts the rows from that. In some cases this can be reduced to a simpler `S c F t`, however modern versions of SQLAlchemy don't try to guess when this is appropriate, as the exact SQL can be emitted using more explicit means.

For situations where the "thing to be counted" needs to be indicated specifically, we can specify the "count" function directly using the expression `f`, available from the `f` construct. Below we use it to return the count of each distinct user name:

Counting on `c`

`Q` used to be a very complicated method when it would try to guess whether or not a subquery was needed around the existing query, and in some exotic cases it wouldn't do the right thing. Now that it uses a simple subquery every time, it's only two lines long and always returns the right answer. Use `f` if a particular statement absolutely cannot tolerate the subquery being present.

```
>>> from sqlalchemy import func
>>> session.query(func.count(User.name), User.name).group_by(User.name).all()
[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

SQL

To achieve our simple `S c F t`, we can apply it as:

```
>>> session.query(func.count('*')).select_from(User).scalar()
4
```

SQL

The usage of `s` can be removed if we express the count in terms of the `U` primary key directly:

```
>>> session.query(func.count(User.id)).scalar()
4
```

SQL

Building a Relationship

Let's consider how a second table, related to `U`, can be mapped and queried. Users in our system can store any number of email addresses associated with their username. This implies a basic one to many association from the `u` to a new table which stores email addresses, which we will call `a`. Using declarative, we define this table along with its mapped class, `A`:

```
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy.orm import relationship

>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...
...     user = relationship("User", back_populates="addresses")
...
...     def __repr__(self):
...         return "<Address(email_address='%s')>" % self.email_address

>>> User.addresses = relationship(
...     "Address", order_by=Address.id, back_populates="user")
```

The above class introduces the `F` construct, which is a directive applied to `C` that indicates that values in this column should be constrained to be values present in the named remote column. This is a core feature of relational databases, and is the "glue" that transforms an otherwise unconnected collection of tables to have rich overlapping relationships. The `F` above expresses that values in the `a` column should be constrained to those values in the `u` column, i.e. its primary key.

A second directive, known as `r`, tells the ORM that the `A` class itself should be linked to the `U` class, using the attribute `A`. `r` uses the foreign key relationships between the two tables to determine the nature of this linkage, determining that `A` will be many to one. An additional `r` directive is placed on the `U` mapped class under the attribute `U`. In both `r` directives, the parameter `r` is assigned to refer to the complementary attribute names; by doing so, each `r` can make intelligent decision about the same relationship as expressed in reverse; on one side, `A` refers to a `U` instance, and on the other side, `U` refers to a list of `A` instances.

> **Note**
>
> The `r` parameter is a newer version of a very common SQLAlchemy feature called `r`. The `r` parameter hasn't gone anywhere and will always remain available! The `r` is the same thing, except a little more verbose and easier to manipulate. For an overview of the entire topic, see the section Linking Relationships with Backref.

The reverse side of a many-to-one relationship is always one to many. A full catalog of available `r` configurations is at Basic Relationship Patterns.

The two complementing relationships `A` and `U` are referred to as a bidirectional relationship, and is a key feature of the SQLAlchemy ORM. The section Linking Relationships with Backref discusses the "backref" feature in detail.

Arguments to `r` which concern the remote class can be specified using strings, assuming the Declarative system is in use. Once all mappings are complete, these strings are evaluated as Python expressions in order to produce the actual argument, in the above case the `U` class. The names which are allowed during this evaluation include, among other things, the names of all classes which have been created in terms of the declared base.

See the docstring for `r` for more detail on argument style.

> **Did you know ?**
>
> - a FOREIGN KEY constraint in most (though not all) relational databases can only link to a primary key column, or a column that has a UNIQUE constraint.
> - a FOREIGN KEY constraint that refers to a multiple column primary key, and itself has multiple columns, is known as a "composite foreign key". It can also reference a subset of those columns.
> - FOREIGN KEY columns can automatically update themselves, in response to a change in the referenced column or row. This is known as the CASCADE *referential action*, and is a built in function of the relational database.
> - FOREIGN KEY can refer to its own table. This is referred to as a "self-referential" foreign key.
> - Read more about foreign keys at Foreign Key - Wikipedia.

We'll need to create the `a` table in the database, so we will issue another CREATE from our metadata, which will skip over tables which have already been created:

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

Working with Related Objects

Now when we create a `U`, a blank `a` collection will be present. Various collection types, such as sets and dictionaries, are possible here (see Customizing Collection Access for details), but by default, the collection is a Python list.

```
>>> jack = User(name='jack', fullname='Jack Bean', password='gjffdd')
>>> jack.addresses
[]
```

We are free to add `A` objects on our `U` object. In this case we just assign a full list directly:

```
>>> jack.addresses = [
```

```
...                    Address(email_address='jack@google.com'),
...                    Address(email_address='j25@yahoo.com')]
```

When using a bidirectional relationship, elements added in one direction automatically become visible in the other direction. This behavior occurs based on attribute on-change events and is evaluated in Python, without using any SQL:

```
>>> jack.addresses[1]
<Address(email_address='j25@yahoo.com')>

>>> jack.addresses[1].user
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

Let's add and commit `J B` to the database. `j` as well as the two `A` members in the corresponding `a` collection are both added to the session at once, using a process known as **cascading**:

```
>>> session.add(jack)
>>> session.commit()
```

Querying for Jack, we get just Jack back. No SQL is yet issued for Jack's addresses:

```
>>> jack = session.query(User).\
... filter_by(name='jack').one()
>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

Let's look at the `a` collection. Watch the SQL:

```
>>> jack.addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

When we accessed the `a` collection, SQL was suddenly issued. This is an example of a lazy loading relationship. The `a` collection is now loaded and behaves just like an ordinary list. We'll cover ways to optimize the loading of this collection in a bit.

Querying with Joins

Now that we have two tables, we can show some more features of `Q`, specifically how to create queries that deal with both tables at the same time. The Wikipedia page on SQL JOIN offers a good introduction to join techniques, several of which we'll illustrate here.

To construct a simple implicit join between `U` and `A`, we can use `Q` to equate their related columns together. Below we load the `U` and `A` entities at once using this method:

```
>>> for u, a in session.query(User, Address).\
...                     filter(User.id==Address.user_id).\
...                     filter(Address.email_address=='jack@google.com').\
...                     all():
...     print(u)
...     print(a)
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
<Address(email_address='jack@google.com')>
```

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the `Q` method:

```
>>> session.query(User).join(Address).\
...         filter(Address.email_address=='jack@google.com').\
...         all()
[<User(name='jack', fullname='Jack Bean', password='gjffdd')>]
```

`Q` knows how to join between `U` and `A` because there's only one foreign key between them. If there were no foreign keys, or several, `Q` works better when one of the following forms are used:

```
query.join(Address, User.id==Address.user_id)     # explicit condition
query.join(User.addresses)                        # specify relationship from left to ri
query.join(Address, User.addresses)               # same, with explicit target
query.join('addresses')                           # same, using a string
```

As you would expect, the same idea is used for “outer” joins, using the `o` function:

```
query.outerjoin(User.addresses)    # LEFT OUTER JOIN
```

The reference documentation for `j` contains detailed information and examples of the calling styles accepted by this method; `j` is an important method at the center of usage for any SQL-fluent application.

What does `Q` select from if there’s multiple entities?

The `Q` method will **typically join from the leftmost item** in the list of entities, when the ON clause is omitted, or if the ON clause is a plain SQL expression. To control the first entity in the list of JOINs, use the `Q` method:

```
query = Session.query(User, Address).select_from(Address).join(User)
```

Using Aliases

When querying across multiple tables, if the same table needs to be referenced more than once, SQL typically requires that the table be *aliased* with another name, so that it can be distinguished against other occurrences of that table. The `Q` supports this most explicitly using the `a` construct. Below we join to the `A` entity twice, to locate a user who has two distinct email addresses at the same time:

SQL

```
>>> from sqlalchemy.orm import aliased
>>> adalias1 = aliased(Address)
>>> adalias2 = aliased(Address)
>>> for username, email1, email2 in \
...     session.query(User.name, adalias1.email_address, adalias2.email_address).\
...     join(adalias1, User.addresses).\
...     join(adalias2, User.addresses).\
...     filter(adalias1.email_address=='jack@google.com').\
...     filter(adalias2.email_address=='j25@yahoo.com'):
...     print(username, email1, email2)
jack jack@google.com j25@yahoo.com
```

Using Subqueries

The `Q` is suitable for generating statements which can be used as subqueries. Suppose we wanted to load `U` objects along with a count of how many `A` records each user has. The best way to generate SQL like this is to get the count of addresses grouped by user ids, and JOIN to the parent. In this case we use a LEFT OUTER JOIN so that we get rows back for those users who don’t have any addresses, e.g.:

```
SELECT users.*, adr_count.address_count FROM users LEFT OUTER JOIN
    (SELECT user_id, count(*) AS address_count
        FROM addresses GROUP BY user_id) AS adr_count
    ON users.id=adr_count.user_id
```

Using the `Q`, we build a statement like this from the inside out. The `s` accessor returns a SQL expression representing the statement generated by a particular `Q` - this is an instance of a `s` construct, which are described in SQL Expression Language Tutorial:

```
>>> from sqlalchemy.sql import func
>>> stmt = session.query(Address.user_id, func.count('*').\
...         label('address_count')).\
...         group_by(Address.user_id).subquery()
```

The `f` keyword generates SQL functions, and the `s` method on `Q` produces a SQL expression construct representing a SELECT statement embedded within an alias (it’s actually shorthand for `q`).

Once we have our statement, it behaves like a `T` construct, such as the one we created for `u` at the start of this tutorial. The columns on the statement are accessible through an attribute called `c`:

SQL

```
>>> for u, count in session.query(User, stmt.c.address_count).\
...     outerjoin(stmt, User.id==stmt.c.user_id).order_by(User.id):
...     print(u, count)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> None
<User(name='wendy', fullname='Wendy Williams', password='foobar')> None
<User(name='mary', fullname='Mary Contrary', password='xxg527')> None
<User(name='fred', fullname='Fred Flinstone', password='blah')> None
<User(name='jack', fullname='Jack Bean', password='gjffdd')> 2
```

Selecting Entities from Subqueries

Above, we just selected a result that included a column from a subquery. What if we wanted our subquery to map to an entity ? For this we use `a` to associate an "alias" of a mapped class to a subquery:

SQL

```
>>> stmt = session.query(Address).\
...                 filter(Address.email_address != 'j25@yahoo.com').\
...                 subquery()
>>> adalias = aliased(Address, stmt)
>>> for user, address in session.query(User, adalias).\
...         join(adalias, User.addresses):
...     print(user)
...     print(address)
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
<Address(email_address='jack@google.com')>
```

Using EXISTS

The EXISTS keyword in SQL is a boolean operator which returns True if the given expression contains any rows. It may be used in many scenarios in place of joins, and is also useful for locating rows which do not have a corresponding row in a related table.

There is an explicit EXISTS construct, which looks like this:

SQL

```
>>> from sqlalchemy.sql import exists
>>> stmt = exists().where(Address.user_id==User.id)
>>> for name, in session.query(User.name).filter(stmt):
...     print(name)
jack
```

The `Q` features several operators which make usage of EXISTS automatically. Above, the statement can be expressed along the `U` relationship using `a`:

SQL

```
>>> for name, in session.query(User.name).\
...         filter(User.addresses.any()):
...     print(name)
jack
```

`a` takes criterion as well, to limit the rows matched:

SQL

```
>>> for name, in session.query(User.name).\
...     filter(User.addresses.any(Address.email_address.like('%google%'))):
...     print(name)
jack
```

`h` is the same operator as `a` for many-to-one relationships (note the `~` operator here too, which means "NOT"):

SQL

```
>>> session.query(Address).\
...         filter(~Address.user.has(User.name=='jack')).all()
[]
```

Common Relationship Operators

Here's all the operators which build on relationships - each one is linked to its API documentation which includes full details on usage and behavior:

- `_` (many-to-one “equals” comparison):

```
query.filter(Address.user == someuser)
```

- `_` (many-to-one “not equals” comparison):

```
query.filter(Address.user != someuser)
```

- IS NULL (many-to-one comparison, also uses `_`):

```
query.filter(Address.user == None)
```

- `c` (used for one-to-many collections):

```
query.filter(User.addresses.contains(someaddress))
```

- `a` (used for collections):

```
query.filter(User.addresses.any(Address.email_address == 'bar'))

# also takes keyword arguments:
query.filter(User.addresses.any(email_address='bar'))
```

- `h` (used for scalar references):

```
query.filter(Address.user.has(name='ed'))
```

- `Q` (used for any relationship):

```
session.query(Address).with_parent(someuser, 'addresses')
```

Eager Loading

Recall earlier that we illustrated a lazy loading operation, when we accessed the `U` collection of a `U` and SQL was emitted. If you want to reduce the number of queries (dramatically, in many cases), we can apply an eager load to the query operation. SQLAlchemy offers three types of eager loading, two of which are automatic, and a third which involves custom criterion. All three are usually invoked via functions known as query options which give additional instructions to the `Q` on how we would like various attributes to be loaded, via the `Q` method.

Subquery Load

In this case we’d like to indicate that `U` should load eagerly. A good choice for loading a set of objects as well as their related collections is the `o` option, which emits a second SELECT statement that fully loads the collections associated with the results just loaded. The name “subquery” originates from the fact that the SELECT statement constructed directly via the `Q` is re-used, embedded as a subquery into a SELECT against the related table. This is a little elaborate but very easy to use:

```
>>> from sqlalchemy.orm import subqueryload
>>> jack = session.query(User).\
...                  options(subqueryload(User.addresses)).\
...                  filter_by(name='jack').one()
>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>

>>> jack.addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

SQL

Note

`s` when used in conjunction with limiting such as `Q`, `Q` or `Q` should also include `Q` on a unique column in order to ensure correct results. See The Importance of Ordering.

Joined Load

The other automatic eager loading function is more well known and is called `o`. This style of loading emits a JOIN, by default a LEFT OUTER JOIN, so that the lead object as well as the related object or collection is loaded in one step. We illustrate loading the same `a` collection in this way - note that even though the `U` collection on `j` is actually populated right now, the query will emit the extra join regardless:

```
>>> from sqlalchemy.orm import joinedload

>>> jack = session.query(User).\
...                        options(joinedload(User.addresses)).\
...                        filter_by(name='jack').one()
>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>

>>> jack.addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

SQL

Note that even though the OUTER JOIN resulted in two rows, we still only got one instance of `U` back. This is because `Q` applies a "uniquing" strategy, based on object identity, to the returned entities. This is specifically so that joined eager loading can be applied without affecting the query results.

While `j` has been around for a long time, `s` is a newer form of eager loading. `s` tends to be more appropriate for loading related collections while `j` tends to be better suited for many-to-one relationships, due to the fact that only one row is loaded for both the lead and the related object.

`j` is not a replacement for `j`

The join created by `j` is anonymously aliased such that it **does not affect the query results**. An `Q` or `Q` call **cannot** reference these aliased tables - so-called "user space" joins are constructed using `Q`. The rationale for this is that `j` is only applied in order to affect how related objects or collections are loaded as an optimizing detail - it can be added or removed with no impact on actual results. See the section The Zen of Eager Loading for a detailed description of how this is used.

Explicit Join + Eagerload

A third style of eager loading is when we are constructing a JOIN explicitly in order to locate the primary rows, and would like to additionally apply the extra table to a related object or collection on the primary object. This feature is supplied via the `o` function, and is most typically useful for pre-loading the many-to-one object on a query that needs to filter on that same object. Below we illustrate loading an `A` row as well as the related `U` object, filtering on the `U` named "jack" and using `o` to apply the "user" columns to the `A` attribute:

```
>>> from sqlalchemy.orm import contains_eager
>>> jacks_addresses = session.query(Address).\
...                             join(Address.user).\
...                             filter(User.name=='jack').\
...                             options(contains_eager(Address.user)).\
...                             all()
>>> jacks_addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]

>>> jacks_addresses[0].user
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

SQL

For more information on eager loading, including how to configure various forms of loading by default, see the section *Relationship Loading Techniques*.

Deleting

Let's try to delete `j` and see how that goes. We'll mark as deleted in the session, then we'll issue a `c` query to see that no rows remain:

```
>>> session.delete(jack)
>>> session.query(User).filter_by(name='jack').count()
0
```

So far, so good. How about Jack's `A` objects ?

```
>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
...  ).count()
2
```

Uh oh, they're still there ! Analyzing the flush SQL, we can see that the `u` column of each address was set to NULL, but the rows weren't deleted. SQLAlchemy doesn't assume that deletes cascade, you have to tell it to do so.

Configuring delete/delete-orphan Cascade

We will configure **cascade** options on the `U` relationship to change the behavior. While SQLAlchemy allows you to add new attributes and relationships to mappings at any point in time, in this case the existing relationship needs to be removed, so we need to tear down the mappings completely and start again - we'll close the `S`:

```
>>> session.close()
ROLLBACK
```

and use a new `d`:

```
>>> Base = declarative_base()
```

Next we'll declare the `U` class, adding in the `a` relationship including the cascade configuration (we'll leave the constructor out too):

```
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
...     addresses = relationship("Address", back_populates='user',
...                     cascade="all, delete, delete-orphan")
...
...     def __repr__(self):
...        return "<User(name='%s', fullname='%s', password='%s')>" % (
...                                 self.name, self.fullname, self.password)
```

Then we recreate `A`, noting that in this case we've created the `A` relationship via the `U` class already:

```
>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     user = relationship("User", back_populates="addresses")
...
...     def __repr__(self):
...         return "<Address(email_address='%s')>" % self.email_address
```

Now when we load the user `j` (below using `g`, which loads by primary key), removing an address from the corresponding `a` collection will result in that `A` being deleted:

```
# load Jack by primary key
>>> jack = session.query(User).get(5)

# remove one Address (lazy load fires off)
```

```
>>> del jack.addresses[1]

# only one address remains
>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
1
```

Deleting Jack will delete both Jack and the remaining `A` associated with the user:

```
>>> session.delete(jack)

>>> session.query(User).filter_by(name='jack').count()
0

>>> session.query(Address).filter(
...    Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
0
```

More on Cascades

Further detail on configuration of cascades is at Cascades. The cascade functionality can also integrate smoothly with the `O D C` functionality of the relational database. See Using Passive Deletes for details.

Building a Many To Many Relationship

We're moving into the bonus round here, but lets show off a many-to-many relationship. We'll sneak in some other features too, just to take a tour. We'll make our application a blog application, where users can write `B` items, which have `K` items associated with them.

For a plain many-to-many, we need to create an un-mapped `T` construct to serve as the association table. This looks like the following:

```
>>> from sqlalchemy import Table, Text
>>> # association table
>>> post_keywords = Table('post_keywords', Base.metadata,
...     Column('post_id', ForeignKey('posts.id'), primary_key=True),
...     Column('keyword_id', ForeignKey('keywords.id'), primary_key=True)
... )
```

Above, we can see declaring a `T` directly is a little different than declaring a mapped class. `T` is a constructor function, so each individual `C` argument is separated by a comma. The `C` object is also given its name explicitly, rather than it being taken from an assigned attribute name.

Next we define `B` and `K`, using complementary `r` constructs, each referring to the `p` table as an association table:

```
>>> class BlogPost(Base):
...     __tablename__ = 'posts'
...
...     id = Column(Integer, primary_key=True)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     headline = Column(String(255), nullable=False)
...     body = Column(Text)
...
...     # many to many BlogPost<->Keyword
...     keywords = relationship('Keyword',
...                             secondary=post_keywords,
...                             back_populates='posts')
...
...     def __init__(self, headline, body, author):
...         self.author = author
...         self.headline = headline
...         self.body = body
...
...     def __repr__(self):
...         return "BlogPost(%r, %r, %r)" % (self.headline, self.body, self.author)
```

```
>>> class Keyword(Base):
...     __tablename__ = 'keywords'
...
...     id = Column(Integer, primary_key=True)
...     keyword = Column(String(50), nullable=False, unique=True)
...     posts = relationship('BlogPost',
...                          secondary=post_keywords,
...                          back_populates='keywords')
...
...     def __init__(self, keyword):
...         self.keyword = keyword
```

Note

The above class declarations illustrate explicit `_` methods. Remember, when using Declarative, it's optional!

Above, the many-to-many relationship is `B`. The defining feature of a many-to-many relationship is the `s` keyword argument which references a `T` object representing the association table. This table only contains columns which reference the two sides of the relationship; if it has *any* other columns, such as its own primary key, or foreign keys to other tables, SQLAlchemy requires a different usage pattern called the "association object", described at Association Object.

We would also like our `B` class to have an `a` field. We will add this as another bidirectional relationship, except one issue we'll have is that a single user might have lots of blog posts. When we access `U`, we'd like to be able to filter results further so as not to load the entire collection. For this we use a setting accepted by `r` called `l`, which configures an alternate **loader strategy** on the attribute:

```
>>> BlogPost.author = relationship(User, back_populates="posts")
>>> User.posts = relationship(BlogPost, back_populates="author", lazy="dynamic")
```

Create new tables:

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

Usage is not too different from what we've been doing. Let's give Wendy some blog posts:

```
>>> wendy = session.query(User).\
...                 filter_by(name='wendy').\
...                 one()
>>> post = BlogPost("Wendy's Blog Post", "This is a test", wendy)
>>> session.add(post)
```

We're storing keywords uniquely in the database, but we know that we don't have any yet, so we can just create them:

```
>>> post.keywords.append(Keyword('wendy'))
>>> post.keywords.append(Keyword('firstpost'))
```

We can now look up all blog posts with the keyword 'firstpost'. We'll use the `a` operator to locate "blog posts where any of its keywords has the keyword string 'firstpost'":

```
>>> session.query(BlogPost).\
...             filter(BlogPost.keywords.any(keyword='firstpost')).\
...             all()
[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Wi
```

If we want to look up posts owned by the user `w`, we can tell the query to narrow down to that `U` object as a parent:

```
>>> session.query(BlogPost).\
...             filter(BlogPost.author==wendy).\
...             filter(BlogPost.keywords.any(keyword='firstpost')).\
...             all()
[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Wi
```

Or we can use Wendy's own `p` relationship, which is a "dynamic" relationship, to query straight from there:

```
>>> wendy.posts.\
...         filter(BlogPost.keywords.any(keyword='firstpost')).\
...         all()
[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Wi
```

SQL

Further Reference

Query Reference: query_api_toplevel

Mapper Reference: Mapper Configuration

Relationship Reference: Relationship Configuration

Session Reference: *Using the Session*