02 SQLAlchemy ORM

April 4, 2021

0.1 Overview

- `SQLAlchemy ORM` is what most people think `SQLAlchemy` is
- it provides effecient ways to bind database schema and operations to the same data objects used in your application
- `ORM` is super simple to use and most people do not think about the effiency of their queries

0.1.1 Defining Schema with SQLAlchemy ORM

- SQLAlchemy ORM is focused around user-defined data objects instead of the schema of the underlying database
- in `SQLAlchemy Core` we create a metadata container and then declared a `Table` object associated with the metadata
- in SQLAlchemy ORM we are going to define a class that inherits from a special base class called `declarative_base`
- the `delaractive_base` combines a metadata container and a mapper that maps our class to a database table
- it also maps instances of the class to records in the table if they have been saved

0.1.2 Defining Tables via ORM Classes

A proper class for use with the ORM must do four things - Inherit from the `declarative_base` object - contain `__tablename__` which is the table name to be used in the database - contain one or more attributes that are Column objects - ensure one or more attributes make up a primary key

- defining columns in ORM class is very similar to defining columns in a `Table object`
- the difference is when defining columns in an `ORM` class, we dont have to supply the column name as the first argument to the `Column` constructor
- instead the column name will be set to the name of the class attribute to which it is assigned
- all types from SQLALchemy `Core` also apply here

- we a primary key because the `ORM` has to have a way to uniquely identify and associate an instance of the class with a specific record in the underlying database table

1. creates an instance of the `declarative_base`
2. Inherit from the `Base`
3. defien the table name
4. define an attribute ansd set it to be a primary key

[1]:
```
from sqlalchemy import Table, Column, Integer, Numeric, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base() # 1

class Cookie(Base): # 2
    __tablename__ = 'cookies' # 3

    cookie_id = Column(Integer(), primary_key=True) # 4
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))


Cookie.__table__
```

[1]:
```
Table('cookies', MetaData(), Column('cookie_id', Integer(), table=<cookies>,
primary_key=True, nullable=False), Column('cookie_name', String(length=50),
table=<cookies>), Column('cookie_recipe_url', String(length=255),
table=<cookies>), Column('cookie_sku', String(length=55), table=<cookies>),
Column('quantity', Integer(), table=<cookies>), Column('unit_cost',
Numeric(precision=12, scale=2), table=<cookies>), schema=None)
```

1. here we are making this column required (`nullable=False`) and requiring the values to be unique
2. the default sets this column to the current time if a date isnt specified
3. using `onupdate` here will reset this column to the current time every time any part of the record is updated

[2]:
```
from datetime import datetime
from sqlalchemy import DateTime

class User(Base):
    __tablename__ = 'user'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True) # 1
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now) #2
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.
 ↪now) #3
```

0.1.3 Key, Constraints and Indexes

- using the `SQLAlchemy ORM` we are building classes and not using the table constructor
- in the ORM, these can be added by using the `__table_args__` attribute on our class
- `__table_args__` expects to get a tuple of additional table arguments

[3]:
```
from sqlalchemy import somedatatable, PrimaryKeyConstraint, UniqueConstraint,␣
↪CheckConstraint

def some_data_class():
    class SomeDataClass(Base):
        __tablename__ = 'somedatatable'
        __table_args__ = (ForeignKeyConstraint(['id'], ['other_table.id']),
                          CheckConstraint(unit_cost >= 0.00,
                              name='unit_cost_positive'))
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-3-a35c0d5c4fef> in <module>
----> 1 from sqlalchemy import somedatatable, PrimaryKeyConstraint,␣
↪UniqueConstraint, CheckConstraint
      2
      3 def some_data_class():
      4     class SomeDataClass(Base):
      5         __tablename__ = 'somedatatable'

ImportError: cannot import name 'somedatatable' from 'sqlalchemy' (c:
↪\users\vicktree\appdata\local\programs\python\python39\lib\site-packages\sqla.chemy\__init_
↪py)
```

0.1.4 Relationships

- the `ORM` uses a similar `ForeignKey` column to constrain and link the objects
- howerver it also uses a `relationship` directive to provide a property that can be used to access the related objects
- the additional propertied that can be accessed on objects has draw backs due to the overhead, but the utility outweight the drawbacks

1. Notice how we import the relationship and backref methods from `sqlalchemy.orm`
2. we are defining a `ForeignKey` just as we did with SQLAlchemy Code
3. this establishes a one-to-many relationsip

[ ]:
```
from sqlalchemy import ForeignKey, Boolean
from sqlalchemy.orm import relationship, backref # 3

class Order(Base):
    __tablename__ = 'orders'
```

```
    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id')) # 2
    shipped = Column(Boolean(), default=False)

    order_by = None

    '''
      what ever you class you give to relationship(), that class will get a
      column applied to it
    '''
    use = relationship("User", backref=backref('orders', order_by=order_by)) # 3
```

- the user relationship defined in the `Order` class establishes a one-to-many relationship with the `User` class
- we can get the `User` related to this `Order` by accessing the user property
- this relationship also establishes an `orders` property on the `User` class via the `backref` keyword argument
- the `relationship` directive needs a target class for the relationship and can optionally include a back reference to be added to target class
- SQLAlchemy knows to use the `ForeignKey` we defined that matches the class we defined in the relationship

- `ForeignKey(users.user_id)` has the `users` table's `user_id` column, maps to the `User` class via the `__tablename__` attribute of `users` and forms the relationship
- it is also possible to establish a one-to-one relationship with the `Cookie` class
- the `uselist=False` keyword argument defines it as a one-to-one relationship
- we also use a simpler back refrence as we do not care to control the order

[ ]:
```
class LineItem(Base):
    __tablename__ = 'line_items'

    line_item_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    # order_by
    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))

    # this establishes a one-to-one relationsip
    cookie = relationship("Cookie", uselist=False)
```

0.1.5 Persisting the Schema

- to create our database tables, we are going to use the `create_all` method on the metadata within our `Base` instance

[ ]:
```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///:memory')

Base.metadata.create_all(engine)
```

0.2 Working with Data via SQLAlchemy ORM

0.3 Session

- the session is the way the `SQLAlchemy ORM` interacts with the database
- it wraps the database connection via an engine, provides an identity map for objects that you load via the session or associate with the session
- the identity map is a cache-like data structure that contains a unique list of objects determined by the objects table and primary key
- a session also wraps a `transaction` and that transcation will be open untill the session is commited or rolled back

- to create a new session, SQLAlchemy provides the `sessionmaker` class to ensure that sessions can be created with the same parameters through an application
- it does this by creating a `Session` class that has been configured according to the arguments passed to the `sessionmaker` factory
- the `sessionmaker` factory should be used just once in your application global scope and treated like a configuration setting

1. Imported the `sessionmaker`
2. Defines a Session class with the bind configuration supplied by `sessionmaker`
3. Create a `session` for our use from our generated `Session` class

[ ]:
```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker # 1

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

Session = sessionmaker(bind=engine) # 2

session = Session() # 3
```

- `session` has everything it needs to connect to the database, it wont connect untill we give it some instructions that require it to do so

[ ]:
```
from datetime import datetime
from sqlalchemy import (Table, Column, Integer, Numeric, String, DateTime,
 ForeignKey)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, backref

Base = declarative_base()

class Cookie(Base):
```

```
    __tablename__ = 'cookies'
    cookie_id = Column(Integer(), primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
                    "cookie_recipe_url='{self.cookie_recipe_url}', " \
                    "cookie_sku='{self.cookie_sku}', " \
                    "quantity={self.quantity}, " \
                    "unit_cost={self.unit_cost})".format(self=self)


class User(Base):
    __tablename__ = 'users'
    user_id = Column(Integer(), primary_key=True)

    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now,
    onupdate=datetime.now)

def __repr__(self):
    return "User(username='{self.username}', " \
        "email_address='{self.email_address}', " \
        "phone='{self.phone}', " \
        "password='{self.password}')".format(self=self)


class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True)
    user_id = Column(Integer(), ForeignKey('users.user_id'))

    user = relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
            "shipped={self.shipped})".format(self=self)

class LineItems(Base):
```

```
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True)
    order_id = Column(Integer(), ForeignKey('orders.order_id'))
    cookie_id = Column(Integer(), ForeignKey('cookies.cookie_id'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))
    order = relationship("Order", backref=backref('line_items',
    order_by=line_item_id))
    cookie = relationship("Cookie", uselist=False, order_by=id)

    def __repr__(self):
            return "LineItems(order_id={self.order_id}, " \
                        "cookie_id={self.cookie_id}, " \
                        "quantity={self.quantity}, " \
                        "extended_cost={self.extended_cost})".format(
                    self=self)


# creeates the table in the database defined by the engine
Base.metadata.create_all(engine)
```

0.3.1 Inserting Data

- to create a new cookie record in our database, we initialize a new instance of the `Cookie` class that has the sdesired data in it
- we then add that new instance of the `Cookie` object to the session and commit the session
- this is even easier to do because inheriting from the `declarative_base` provides a default constructor that we use

1. creating an instance of the `Cookie` class
2. Adding the instance to the session
3. Commiting the session

[ ]:
```
def insert():
    cc_cookie = Cookie(cookie_name='chocolate chip',
                cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                cookie_sku='CC01',
                quantity=12,
                unit_cost=0.50)

    session.add(cc_cookie) # 2
    session.commit() # 3



    #print(cc_cookie.cookie_id)
```

when running insert, the following happens in the database: - we create the instance of the `Cookie` class and then add it to the session, nothing is sent to the database - when we call`commit()` on the

session, then eveything is sent to the database

- when `commit` is called, the following happens

```
# start transcation
INFO:sqlalchemy.engine.base.Engine:BEGIN (implicit)

# insert the record into the database
INFO:sqlalchemy.engine.base.Engine:INSERT INTO cookies  (cookie_name, cookie_recipe_url, cookie

# the values for the insert
INFO:sqlalchemy.engine.base.Engine:('chocolate chip', 'http://some.aweso.me/cookie/recipe.html

# commit the transcation
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

- to get the details of whats happening in your datbase, all you have do to do is set `echo=True` in the `create_engine` statement

1. adds the dark choclate chip cookie
2. adds the molasses cookie
3. flushes the session

[ ]:
```
def insert_multiple():
    dcc = Cookie(cookie_name='dark chocolate chip',
        cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
        cookie_sku='CC02',
        quantity=1,
     unit_cost=0.75)

    mol = Cookie(cookie_name='molasses',
        cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
        cookie_sku='MOL01',
        quantity=1,
        unit_cost=0.80)

    session.add(dcc) # 1
    session.add(mol) # 2
    session.flush()  # 3

    print(dcc.cookie_id)
    print(mol.cookie_id)
```

- a `flush()` is like a commit, howerver, it doesnt perform a database commit and end the transaction
- because of this, instances are still connected to the session and can be used to perform additional database tasks without triggering additional database queries
- we also issue the `session.flush()` statement one time even though we added multiple records into the database
-

[ ]:
```
def bulk_save():
    c1 = Cookie(cookie_name='peanut butter',
        cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
        cookie_sku='PB01',
        quantity=24,
        unit_cost=0.25)
    c2 = Cookie(cookie_name='oatmeal raisin',
        cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
        cookie_sku='EWW01',
        quantity=100,
        unit_cost=1.00)

    # notice we are not using session.add()
    # adds the cookies to a list and saves them all
    session.bulk_save_objects([c1,c2])
    session.commit()
    print(c1.cookie_id)
```

the speed of using `bulk_save_object` comes at the expense of - relationship settings and actions are not respected or triggered - the objects are not conencted to the session - fetching primary keys is not done by default - no events will be triggered

> use `bulk_save_objects` when you are ingesting data from an external data source such as a `CSV` or a large `JSON` document with nested arrays

0.3.2 Querying Data

- to build a query you start by using the `query()` method on the session instance
- the return value is a of objects
- these objects are connected to the session, which menas we can change them or delete them and persist that change to the database

[ ]:
```
cookies = session.query(Cookie).all()
```

- if we wnat to iterater throug all of them, we dont append `all()`

[ ]:
```
for cookie in session.query(Cookie):
    print(cookie)
```

- there are a few other methods to fetch results
- `first()`
  - returns the first object if there is one
- `one()`
  - queries all the rows and raises an exception if anything other than a single result is returned
- `scalar()`
  - returns the first element of the first result, `None` if there is no result, or an error if there is more than one result

Tips for Good Production Code: - use the iterable version of the query over the `all()` method.

It is more memory efficient than handling a full list of objects and we tend to operate on the data one record at a time anyway - to get a single record, use the `first()` method (rather than `one()` or `scalar()` because it is clearer to developers. The only exception to this is when you must ensure that there is one and only one result from a query - use the `scalar()` method sparingly; it raises errors if a query ever returns more than one row with one column. In a query that selects entire records, it will return the entire record object which is confusing

- sqlalchemy does not add a bunch of overhead to the queries or objects; however, accounting for the data you get back from a query is often the first place to look if a query is consuming too much memory

0.3.3 Controlling the Columns in the Query

Ordering: - use the `order_by()` statement - `sessions.query(Cookie).order_by(Cookie.quantity)` - use `desc()` to reverse order - you can use `desc` as a function or method but it is perfered you use it as a `function`

Limiting: - we can use array slicing - `session.query(Cookie).order_by(Cookie.quantity)[:2]` - slicing can be very ineffecient with large result set - we also have the `limit()` statement - `...limit(2)`

0.3.4 Built-in SQL Functions and Labels

- to use SQL function in the backend database we need to import `sqlalchemy.func` module generator that makes them avaliable
- `inv_count = session.query(func.sum(Cookie.quantity)).scalar()`
- `....(func.count(Cookie.cookie_name)..`
- using functions such as `count()` and `sum()` will end up returning tuples or results with column names like `count_1`
- SQLAlchemy provides a way to lable these returns more explicitly with the use of the `label()` function
- `...query(func.count(Cookie.cookie_name).label('inventory_count').first()`

0.3.5 Filtering

- filtering queries is done by appending `filter()` statement to our query
- a typical filter() clause has a `column`, an `operator` and a `value` or `column`
- you can chain multiple `ClauseElement` expressions in a filter statement with the use of `ANDs`
- `...query(Cookies).filter(Cookie.cookie_name == 'choclate chip').first()`
- `filter_by()` methods works similarly to the `filter()` method except instrad of explicitly providing the class as part of the filter expression it uses attribute keyword expressions from the primary entity of the query or the last entity that was joined to the statement
- `...filter_by(cookie_name='choclate chip').first()`
- we can also use a where statement to find all the cookie names that contain the work “choclate”
- `filter(Cookie.cookie_name.like("%chocolate%"))`
- there are negative versions like `notlike` and `notin_()`
- if we do not use these `ClauseElements` we will have to use operators

Table 2-1. ClauseElement methods

Method	Purpose
`between(cleft, cright)`	Find where the column is between `cleft` and `cright`
`concat(column_two)`	Concatenate column with `column_two`
`distinct()`	Find only unique values for the column
`in_([list])`	Find where the column is in the list
`is_(None)`	Find where the column is `None` (commonly used for Null checks with None)
`contains(string)`	Find where the column has *`string`* in it (case-sensitive)
`endswith(string)`	Find where the column ends with *`string`* (case-sensitive)
`like(string)`	Find where the column is like *`string`* (case-sensitive)
`startswith(string)`	Find where the column begins with *`string`* (case-sensitive)
`ilike(string)`	Find where the column is like *`string`* (this is *not* case-sensitive)

0.3.6 Operators

- we got all of the python operators
- the `==` operator gets an additional overload when compared to `None`. which converts it to an `IS NULL` statement
- you got you classic string concatnation
- you can also use `operators` to compute values from multiple columns, where you can multiply them, ett
- `cast(Cookie.quantity * Cookie.unit_cost)` -cast`is a function that allows us to convert types, simmilar to`.fomat- `note without the use of`label‘, the column would not be listed in the keys of the result object

0.3.7 Boolean Operators

- you want to be careful with using bit wise operators such as `&`, `|`, `~`
- the bitwise operators take presendance over normal operators

0.3.8 Conjunctions

- we can chain multiple `filter()` clauses together but its not readable
- the conjuctions are `and_()`, `or_()` and `not_()`

0.3.9 Updating Data

- simmilar to the `insert` clause but you can specifiy a `where` clause to specify what you want to update
- you can either get the object, update an attribute and then commit it or you can update it in place
  - `query.update({Cookie.quantity: Cookie.quantity - 20})`
- the `update()` method causes the record to be updated outside of the session and returns the number of rows updated

0.3.10 Deleting Data

- you can use the `delete()` function or the `delete()` method on the table
- `delete` takes no value paramaters, only an `where` clause
- `session.delete(dcc_cookie)`
- we can also do an inplace delete: `query.delete()`

0.3.11 Joins

[ ]:
```
def joins():

    query = session.query(Order.order_id, User.username, Cookie.cookie_name)

    query = query.join(User).join(LineItem).join(Cookie)

    result = query.filter(User.username == 'cookiemon').all()
```

- if we wanted to include stuff from outside of our data, we use the `outerjoin()` method
- we are joining stuff here but what if we have a self-referential table like a table of `managers` and their `reports`?
- the ORM allows us to establish a relationship that points to the same table
- we need to specify `remote_side`

[ ]:
```
def self_referential_table():
    class Employee(Base):
        __tablename__ = 'employees'

        id = Column(Integer(), primary_key=True)
        manager_id = Column(Integer(), ForeignKey('employees.id'))
        name = Column(String(255), nullable=False)

        manager = relationship("Employee", backref=backref('reports',
                                                           remote_side=[id]))
```

0.3.12 Grouping

- when grouping, you need one or more columns to group on and one or more columns that it makes sense to aggregate with `count`, `sum`, `etc.`
- `session.quert(User,username, func.count(Order.order_id))`
- `query.outerjoin(Order).group_by(User.username)`

0.3.13 Chaning

- chaining is powerful if you use it conditionally
- for example, you want to use filter if `X` is true other wise you dont filter

0.3.14 Raw Queries

- sometime you want to use small text snippets to make the query easy to understand
- that is where you would use `text()`
- `session.quert(User).filter(text("username='test'")`

0.4 Understanding the Session and Exceptions

0.4.1 Session States

- there are four possible states for data object instances
- **Transient**:
  - the instance is not in session and is not in the database
- **Pending**:
  - the instance has been added to the session with `add()` but hasent been flushed or commited
- **Persistent**:
  - the object in session has a corresponding record in the database
- **Detached**:
  - the instance is no longer connected to the session, but has a record in the database
- to see the instance state, we can use a powerful `inspect()` method provided by SQLAlchemy from sqlalchemy import inspect inspect = inspect(cc_cookie)
- we can loop through the sesion state to get it for state in [‘transient’, ‘pending’, ‘persistent’, ‘detched’]: print(’{:>10}: {}.format(state, getattr(insp, state)))

```
transient: True
pending: False
persistent: False
detached: False
```

- the state of the cookie is in `transient`, which is the state all newly created objects are in
- if we add `cc_cookie` into the current session and rerun, we get `pending` to be true
- after the cookie has been added, the state moves to `persistent`
- to get the cookie to the `detached` state, we call the `expunge()` method on the session
  - you do this if you are moving data from one session to another

  - an example of this is when you are moving data from your primary database to a data warehouse

- in production mode, you want to call `insp.transient`, `insp.pending`, etc

- we can also see how an object moves through the session states
- we have to use the inspectors modified property: `insp.modified`
- this will return `True` and we can use the inspectors `attrs` collection to find out what changed
  1. checks the attribute state to see if the session can find any changed
  2. prints the `history object` of the changed attribute
     for attr, attr_state in insp.attrs.items(): if attr_state.history_has_changed(): (1) print(f'{attr}: {attr_state.value') print('history: {attr_state.history}' (2)

0.4.2 Exceptions

- two common errors are `MultipleResultsFound` and `DetachedInstanceError`
- intersting thing about thest exceptions is that they tell you which line they are occuring on

MultipleResultsFound Execption

- the exception occurs when we use the `.one()` query method but get more than one result back
- another error related to this one is the `NoResultFound` exception which occurs when you use `.one()` method and the query returned no results

- all of SQLAlchemy exceptions are available in the `sqlalchemy.orm.exc` module

DetachedInstanceError

- occurs when we attempt to access an attribute on an instance that needs to be loaded from the database, but the instance we are using is not currently attached to the database
- this error acts simmilar to `ObjectDeletedError`, `StaleDataError`, `ConcurrentModificationError`
- those errors are related to information differing between the instance, the session, and the database

- in the code below, we are querying to get an instance of an order
- ocne we have our instance, we detach it from the session with expunge
- then we attempt to load the `line_items` attributes
- because `line_items` is a relationship, by default it doesent load all that data untill you ask for it
- in our case, we detached the instance from the session and the relationship doesent have a session to execute a query to load the `line_items` attribute and it raises the 'DettachedInstanceError

[ ]:
```
def triggere_detached_instance_error():

    cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password')
    session.add(cookiemon)
    o1 = Order()
    o1.user = cookiemon
```

```
    session.add(o1)

    cc = session.query(Cookie).filter(Cookie.cookie_name == "Change choclate␣
↪chip").one()

    line1 = LineItem(order=o1, cookie=cc, quantity=2, extended_cost=1.00)

    session.add(line1)
    session.commit()

    # causing the detachment error
    order = session.query(Order).first()
    session.expunge(order)
    order.line_items
```

- to solve this issue, you could use a `try/excpet` block to add it back to the session
- the problem howerver, is that the error is normally an indicator of an exception occuring prior to this point

- all of these exceptions above are due to errors occuring in a single line
- for failures occuring on multiple statements, we need to handle them using transactions

0.4.3 Transactions

- `Transactions` are a group of statements that we need to succeed or fail as a group
- when we first create a session, it is not connected to a database
- when we undertake our first action wih a session such as a `query`, it starts a connection and a transaction
- this means by default we don't need to manually create transactions

- howerver, if we need to handle any exception where part of the transaction succeeds and another part fails or where the result of a transaction creates an exception, then we must know how to control the transaction manually

- an example of this is when we add a constraint such as not completing an order if the number of cookies is below a threshold

```
class Cookie(Base):
    __table_args__ = (CheckConstraint('quantity >=0', name='quantity_postive'),)
```

- with the constraint we basically broke our current session and if we attempt to issue any more statements via the session such as a query, we get the following error:
  - `InvalidRequestError: This Session's transaction has been rolled back due to a previous exception during flush.`
  - the `InvalidRequestError` is really caused due to the `IntergityErro`

- to recover from this session state, we need to manually roll back the transaction

- the `session.rollback()` method on the session will restore our session to a working station

[ ]:
```
from sqlalchemy.exc import IntegrityError

def ship_it(order_id):
    order = session.query(Order).get(order_id)

    for li in order.line_items:
        li.cookie.quantity = li.cookie.quantity - li.quantity
        session.add(li.cookie)

    order.shipped = True
    session.add(order)

    try:
        session.commit()
        print("shipped order ID: {}".format(order_id))
    except IntegrityError as error:
        print('ERROR: {!s}'.format(error.orig))

        # rolling back the transcation if an exception occurs
        session.rollback()
```

0.5 Testing With SQLAlchemy ORM

- in `SQLAlchemy` it can be alot of work to correctly mock out a query statement or a model for unit testing
- so we end up testing against a database and how to mock out `SQLAlchemy` queries and connections

0.5.1 Testing with a Test Database

- out databse is setup via the `DataAccessLayer` class
- we will use that class to initialize the database engine and session whenever we like

[ ]:
```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class DataAccessLayer:

    # initializes a connection with a specific connection string like a factory
    def __init__(self):
        self.engine = None
        self.conn_string = 'some conn string'

    # creates all the tables in out Base class and uses sessionmaker to create
    # a easy way to make sessions for our use in our application
    def connect(self):
        self.engine = create_engine(self.conn_string)
```

```
        Base.metadata.create_all(self.engine)
        self.Session = sessionmaker(bind=self.engine)

# the connect method provides an isntance of the DataAccessLayer class that
# can be imported through the application
dal = DataAccessLayer()

class Cookie(Base):
    pass


class User(Base):
    pass


class LineItem(Base):
    pass


def get_orders_by_customer(cust_name, shipped=None, details=False):
    query = dal.session.query(Order.order_id, User.username, User.phone)
    query = query.join(User)

    if details:
        query = query.add_columns(Cookie.cookie-name, LineItem.quantity,
                                  LineItem.extened_cost)
        query = query.join(LineItem).join(Cookie)

        if shipped is not None:
            query = query.filter(Order.shipped == shipped)
        result = query.filter(User.username == cust_name).all()
        return results
```

- the things we want to test are:
  - `cust_name` which can be blank, a valid or invalid string
  - `shipped` can be `None`, `True` or `False`
  - `details` can be True or False
  - in other words, test all the possible combinations, we will need 12 ($3 \mathcal{2}2$) tests

[ ]:
```
import unittest
# from db import dal

class TestApp(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        dal.conn_string = 'sqllite:///:memory:'
```

```
        dal.connect()
        dal.session = dal.Session()

        # a function that adds data to our session
        prep_db(dal.session)
        dal.session.close()


    # we need to create a new session for every test, that is why
    # we add it in the setup
    def setUp(self):
        dal.session = dal.Session()

    # we have to rollback every session that is why we add that in
    # the teardown
    def tearDown(self):
        dal.session.rollback()
        dal.session.close()

def test_orders_by_customer_blank(self):
    results = get_orders_by_customer('')
    self.assertEqual(results, [])


def test_orders_by_customer_blank_details(self):
    results = get_orders_by_customer('', details=True)
    self.assertEqual(results, [])

'''
def test...
'''
```

0.5.2 Using Mocks

- powerful technique when you have a test enviorment where creating a test database does not make sense or simplyis not feasible
- if you have a large amount of logic that operates on the result of the query, it can be useful to mock out the SQLAlchemy code to return the value you want
- you can still create an `in-memeory` database but just dont load any data into it
- you can also mock out the database connection itself

1. Patching `dal.session` in the app module with a mock
2. that mock is passed into the test function as `mock_dal`
3. we set the return value of the execute method to the chained return value of the all method which we set to `self.cookie_order`
4. now we call the test function where the `dal.connection` will be mocked and return the value we set in the prior step

[ ]:
```
import mock

@mock.patch('app.dal.session') # 1
def test_orders_by_customer(self, mock_dal): # 2
    mock_dal.query.return_value.join.return_value.filter.return_value. \
        all.return_value = self.cookie_orders # 3
    results = get_orders_by_customer('cookiemon') # 4
    self.assertEqual(results, self.cookie_orders)
```

0.6 Relfection with SQLAlchemy ORM and Automap

- if you want to reflect a database schema into `ORM-style` classes we have to use the SQLAlchemy extension automap that lets us do that
- we will be using the `Chinook` database

0.6.1 Reflecting a Database with Automap

1. this reflection has created ORM objects for each table that is accessible under the `class` property of the `automap Base`

[ ]:
```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy import create_engine

# initialize a Base object
Base = automap_base()
engine = ''

try:
    engine = create_engine('sqlite://Chinook_Sqlite.sqlite')
except:
    print('cant create engine Chinook_not exists')


# reflects the entire database
try:
    Base.prepare(engine, reflect=True) # 1
except:
    print('cant reflext databse Chinook_not exists')

try:
    Base.classes.keys
except:
    print('cant get list of classes becasue Chinkook_not exist')

try:
    Artist = Base.classes.Artist
    Album = Base.classes.Album
except:
```

```
    print('cant create objects to refrence Artist and Album becasue Chinook_not␣
↪exist')
```

- we can perform a simply query on the objects that were reflected from the database

```
session = Session(engine)
for artist in session.query(Artist).limit(10:
    print(artist.ArtistId, artist.Name)
```

0.6.2 Reflected Relationships

- automap can automatically reflect and establish `many-to-one`, `one-to-many`, and `many-to-many` relationships
- when automap creates a relationship, it creates a `<related-object>_collection` property on the object

```
artist = session.quert(Artist).first()
for album in artist.album_collection:
    print(f'{artist.name} - {album.Title}')
```

- you can configure automap and override certain aspects of its behavior to tailor the classes it creates to percise sepcifications