SQLAlchemy: Python SQL Toolkit and Object Relational Mapper

http://www.sqlalchemy.org/

Martin Geisler

Rackspace International,
Zurich Switzerland

April 3rd, 2014

Outline

Introduction

The Session

The Mapper

The Magic

Conclusion

the open cloud company

Outline

Introduction

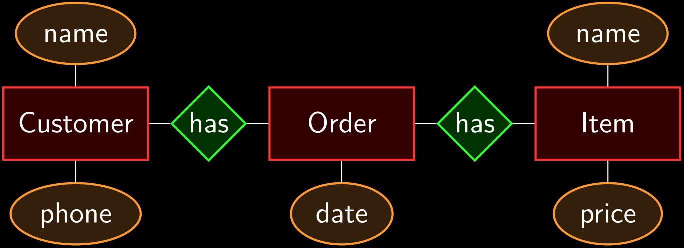
The Session

The Mapper

The Magic

Conclusion

A Very Simple Database



Pure sqlite3 Code I

You start with something like this:

```
def open_db(path):
    conn = sqlite3.connect(path)
    conn.row_factory = sqlite3.Row
    return conn

def create_tables(conn):
    conn.execute("CREATE TABLE Customers (name text, phone text)")
    conn.execute("CREATE TABLE Orders (date text, customerid int)")
    conn.execute("CREATE TABLE OrderItems (orderid int, itemid int)")
    conn.execute("CREATE TABLE Items (name text, price real)")
    conn.commit()
```

Pure sqlite3 Code II

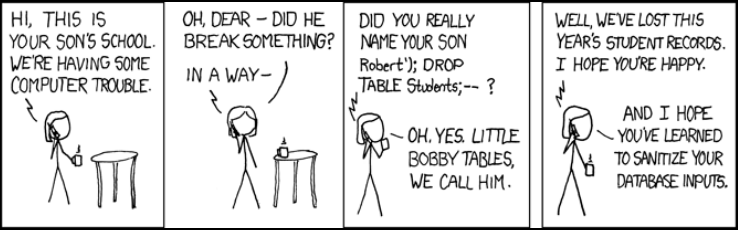
Let us model a Customer:

```python
class Customer:
    def __init__(self, name, phone, rowid=None):
        self.name = name
        self.phone = phone
        self.rowid = rowid
```

Pure sqlite3 Code III

You extend it with a class method to load a Customer:

```
class Customer: # ...
    @classmethod
    def load(cls, conn, rowid):
        cur = conn.execute("SELECT name, phone FROM Customers "
                           "WHERE rowid = ?", (rowid,))
        row = cur.fetchone()
        return cls(row["name"], row["phone"], rowid)
```


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Pure sqlite3 Code IV

You also need to save them:

```
class Customer: # ...
    def save(self, conn):
        cur = conn.cursor()
        if self.rowid is None:
            cur.execute( "INSERT INTO Customers (name, phone) "
                          "VALUES (?, ?)", (self.name, self.phone))
            self.rowid = cur.lastrowid
        else:
            cur.execute("UPDATE Customers SET name = ?, phone = ? "
                         "WHERE rowid = ?",
                         (self.name, self.phone, self.rowid))
        conn.commit()
```

Pure sqlite3 Code V

You need similar code to handle Order objects:

```
class Order:
    def __init__(self, date, customer, rowid=None):
        self.date = date
        self.customer = customer
        self.rowid = rowid
        self.items = []
```

Pure sqlite3 Code VI

Loading an Order looks familiar:

```
class Order: # ...
    @classmethod
    def load(cls, conn, rowid):
        cur = conn.execute("SELECT date, customer "
                           "FROM Orders WHERE rowid = ?",
                           (rowid,))
        row = cur.fetchone()
        return cls(row["date"], Customer.load(row["customer"]), rowid)
```

Pure sqlite3 Code VII

Yet more familiar code:

```
class Order: # ...
    def save(self, conn):
        cur = conn.cursor()
        if self.rowid is None:
            cur.execute("INSERT INTO Orders (date, customerid) "
                        "VALUES (?, ?)",
                        (self.date, self.customer.rowid))
            self.rowid = cur.lastrowid
        else:
            cur.execute("UPDATE Orders SET date = ?, customerid = ? "
                        "WHERE rowid = ?",
                        (self.date, self.customer.rowid, self.rowid))
        conn.commit()
```

the open cloud company

Pure sqlite3 Code VIII

You would like to load orders for a Customer:

```
class Customer: # ...
    def orders(self, conn):
        sql = ("SELECT Orders.rowid, date, Items.rowid, name, price "
               "FROM Orders "
               " JOIN OrderItems ON Orders.rowid = OrderItems.orderid "
               " JOIN Items ON OrderItems.itemid = Items.rowid "
               "WHERE Orders.customerid = ?")
        results = []
        order_rowid = None
        for row in conn.execute(sql, (self.rowid,)):
            if order_rowid != row[0]:
                order = Order(date=row[1], customer=self, rowid=row[0])
                results.append(order)
                order_rowid = row[0]
            order.items.append(Item(row[3], row[4], row[2]))
        return results
```

SQLAlchemy to the Rescue! I

SQLAlchemy reduces the boiler-plate code significantly:

```
class Customer(Base):
    __tablename__ = "Customers"
    rowid = Column(Integer, primary_key=True)
    name = Column(String)
    phone = Column(String)
    orders = relationship("Order")

class Order(Base):
    __tablename__ = "Orders"
    rowid = Column(Integer, primary_key=True)
    date = Column(String)
    customer_id = Column(Integer, ForeignKey("Customers.rowid"))
```

SQLAlchemy to the Rescue! II

The Item table:

```
class Item(Base):
    __tablename__ = "Items"
    rowid = Column(Integer, primary_key=True)
    name = Column(String)
    price = Column(Integer)
```

SQLAlchemy to the Rescue! III

Order with a many-to-many relationship to Item:

```
order_items = Table("OrderItems", Base.metadata,
    Column("order_id", Integer, ForeignKey("Orders.rowid")),
    Column("item_id", Integer, ForeignKey("Items.rowid")))

class Order(Base):
    __tablename__ = "Orders"
    rowid = Column(Integer, primary_key=True)
    date = Column(String)
    customer_id = Column(Integer, ForeignKey("Customers.rowid"))
    items = relationship("Item", secondary=order_items)

class Item(Base):
    __tablename__ = "Items"
    rowid = Column(Integer, primary_key=True)
    name = Column(String)
    price = Column(Integer)
```

SQLAlchemy to the Rescue! IV

Setting up the code:

```
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import create_engine
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Customer(Base): # ...
class Order(Base): # ...
class Item(Base): # ...

if __name__ == "__main__":
    engine = create_engine("sqlite:///:memory:", echo=True)
    Base.metadata.create_all(engine)
```

Overview

SQLAlchemy is a powerful library for working with databases:

- high performance
- mature and well-tested
- great documentation
- high abstraction level
- very feature rich

Features

Many. . . documentation spans 1100 pages!

When starting it's good to know that:

- scales well with your needs
- allows you to insert raw SQL when needed
- can be used with existing databases

Installation

SQLAlchemy is of course on PyPI:

```
$ pip install sqlalchemy
```

Install your favority DBAPI package too:

```
$ pip install psycopg2
```

SQLAlchemy has support for PostgreSQL, MySQL, SQLite, . . .

Outline

Introduction

The Session

The Mapper

The Magic

Conclusion

the open cloud company

The Session

The session is your interface to the DB:

- provides interface for queries
- implements transaction behavior
- maintains an identity map

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

def handle_request(request):
    session = Session()
    try:
        # use the session...
        session.commit()
    except:
        session.rollback()
    session.close()
```

rackspace
the open cloud company

Adding and Deleting Objects

Simply add the object to the Session:

```
c1 = Customer(name="Roy Trenneman", phone="078 123 45 67")
c2 = Customer(name="Maurice Moss", phone="044 551 10 12")
session.add(c1)
session.add(c2)
```

Delete them in the same fashion:

```
session.delete(c1)
```

The changes are written to the DB when the session is committed:

```
session.commit()
```

rackspace

the open cloud company

Querying the Database

Use the Session.query method to query the DB:

```
customers = session.query(Customer).filter_by(name="John Doe")
for customer in customers:
    order_count = len(customer.orders)
    print customer.name, "has", order_count, "orders"
    if order_count > 5:
        customer.name += " The Great"
```

Identity Map

The session keeps track of the DB rows:

```
c1 = session.query(Customer).filter_by(name="John Doe").one()
c2 = session.query(Customer).filter_by(phone="078 123 45 67").one()
assert c1.name == c2.name
assert c1.phone == c2.phone
assert c1 is c2
```

This ensure the application has a consistent view of the DB.

Outline

Introduction

The Session

The Mapper

The Magic

Conclusion

rackspace®
the open cloud company

The Mapper

The Mapper is mostly hidden:

- associates Python classes with DB tables
- decides which Python field corresponds to which DB column
- used indirectly when you create Column objects

Mapping Columns

Many possibilities:

```
class User(Base):
    __tablename__ = "users"
    user_id = Column("id", Integer, primary_key=True)
    firstname = Column(String)
    lastname = Column(String)
    name = column_property(firstname + " " + lastname)
    photo = deferred(Column(Binary))
```

Simple Computed Columns

With

```
class User(Base):
    # ...
    name = column_property(firstname + " " + lastname)
```

you can query on name:

```
jen = session.query(User).filter_by(name="Jen Barber").one()
```

SQLAlchemy will generate this SQL for SQLite:

```
SELECT users.id AS users_id,
  users.firstname || ? || users.lastname AS anon_1,
  users.firstname AS users_firstname,
  users.lastname AS users_lastname
FROM users
WHERE users.firstname || ? || users.lastname = ?
```

Advanced Computed Columns

Generate custom SQL on set and get:

```
class EmailAddress(Base):
    __tablename__ = "email_address"
    id = Column(Integer, primary_key=True)
    _email = Column("email", String)

    @hybrid_property
    def email(self):
        return self._email[:-12]

    @email.setter
    def email(self, email):
        self._email = email + "@example.com"

    @email.expression
    def email(cls):
        return func.substr(cls._email, 0, func.length(cls._email) - 12)
```

Custom Column Types I

Create new column type:

```
class PhoneColumn(TypeDecorator):
    impl = VARCHAR(15)
    number_format = phonenumbers.PhoneNumberFormat.E164

    def process_bind_param(self, value, dialect):
        if not isinstance(value, phonenumbers.PhoneNumber):
            value = phonenumbers.parse(value, "CH")
        return phonenumbers.format_number(value, self.number_format)

    def process_result_value(self, value, dialect):
        return phonenumbers.parse(value)
```

This stores phone numbers +41781234567 in DB.

Custom Column Types II

Use the new column like:

```
class Customer(Base):
    # ...
    phone = PhoneColumn()
```

Query for values based on new column:

```
customer = session.query(Customer).filter_by(phone="078 123 45 67")
print "Country code:", customer.phone.country_code
```

Outline

Introduction

The Session

The Mapper

The Magic

Conclusion

rackspace®

the open cloud company

Magic Fields

How does this really work:

```
customer.phone = '078 234 5678'
```

Fields of mapped classes

- transparently load data from DB
- transparently record modifications
- yet have the correct types (int, str, . . .)

What kind of magic is this?

The Descriptor Protocol I

Python has something called descriptors:

```
class VerboseField(object):
    def __init__(self, name):
        self.name = name
        self.data = weakref.WeakKeyDictionary()
    def __get__(self, instance, owner):
        print '*** reading', self.name
        return self.data[instance]
    def __set__(self, instance, value):
        print '*** setting', self.name, 'to', value
        self.data[instance] = value

class SomeClass(object):
    x = VerboseField()
```

The Descriptor Protocol II

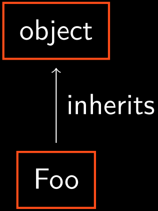
Now try accessing the x field:

```
>>> a = Foo()
>>> a.x
*** reading field
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __get__
  File "/usr/lib/python2.7/weakref.py", line 315, in __getitem__
    return self.data[ref(key)]
KeyError: <weakref at 0x7f2eb2cad158; to 'Foo' at 0x7f2eb2ca4d90>
>>> a.x = 'hello'
*** setting field to hello
>>> a.x
*** reading field
'hello'
```

Metaclasses

Metaclasses allow you to customize class construction:

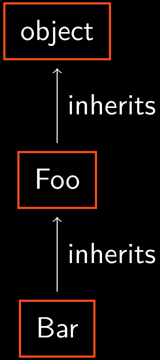
- you can pass classes around in Python — they are objects
- a metaclass is the class invoked to create a “class object”



Metaclasses

Metaclasses allow you to customize class construction:

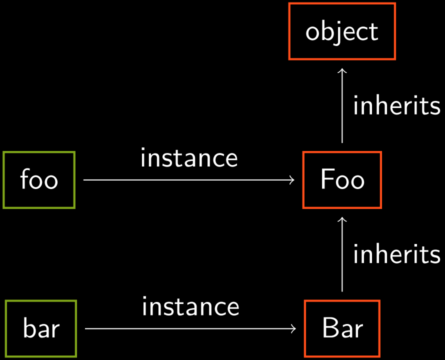
- you can pass classes around in Python — they are objects
- a metaclass is the class invoked to create a “class object”



Metaclasses

Metaclasses allow you to customize class construction:

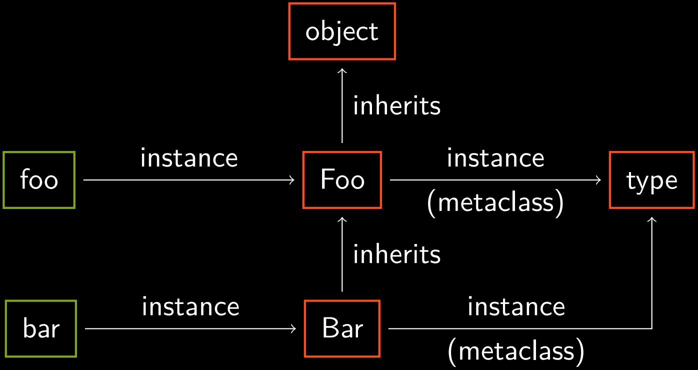
- you can pass classes around in Python — they are objects
- a metaclass is the class invoked to create a “class object”



Metaclasses

Metaclasses allow you to customize class construction:

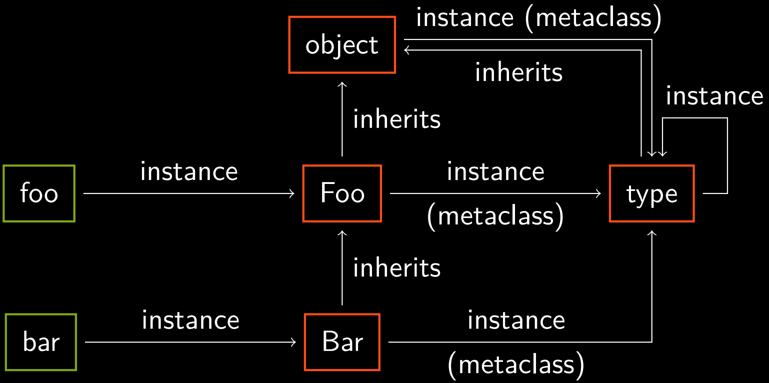
- you can pass classes around in Python — they are objects
- a metaclass is the class invoked to create a "class object"



Metaclasses

Metaclasses allow you to customize class construction:

- you can pass classes around in Python — they are objects
- a metaclass is the class invoked to create a “class object”



Metaclass Example

Simple example of a metaclass that marks everything private:

```python
class MakePrivate(type):
    def __new__(cls, name, bases, attrs):
        private = {}
        for key, value in attrs.iteritems():
            private['_' + key] = value
        return super(MakePrivate, cls).__new__(cls, name, bases, private)

class Foo(object):
    __metaclass__ = MakePrivate
    x = 10

foo = Foo()
print foo._x
```

This yields 10.

Outline

Introduction

The Session

The Mapper

The Magic

Conclusion

rackspace®
the open cloud company

Conclusion

SQLAlchemy makes databases fun again!

- easy things become easy
- difficult things are still possible

the open cloud company

Conclusion

SQLAlchemy makes databases fun again!

- easy things become easy
- difficult things are still possible

Thank you! Questions?