Python's SQLAlchemy and Object-Relational Mapping

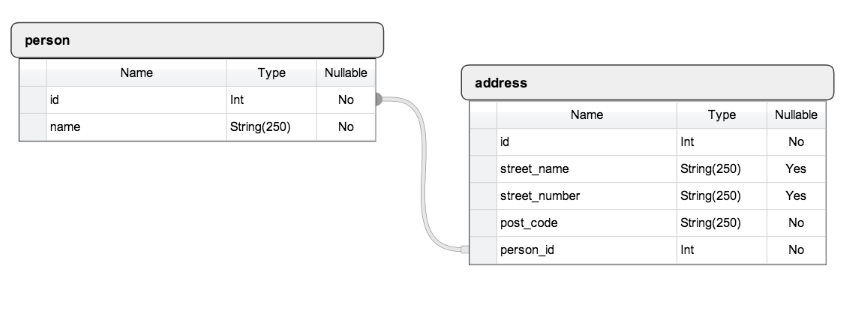
A common task when programming any web service is the construction of a solid database backend. In the past, programmers would write raw SQL statements, pass them to the database engine and parse the returned results as a normal array of records. Nowadays, programmers can write *Object-relational mapping* (*ORM*) programs to remove the necessity of writing tedious and error-prone raw SQL statements that are inflexible and hard-to-maintain.

ORM is a programming technique for converting data between incompatible type systems in object-oriented programming languages. Usually, the type system used in an OO language such as Python contains types that are non-scalar, namely that those types cannot be expressed as primitive types such as integers and strings. For example, a Person object may have a list of Address objects and a list of PhoneNumber objects associated with it. In turn, an Address object may have a PostCode object, a StreetName object and a StreetNumberobject associated with it. Although simple objects such as PostCodes and StreetNames can be expressed as strings, a complex object such as a Address and a Person cannot be expressed using only strings or integers. In addition, these complex objects may also include instance or class methods that cannot be expressed using a type at all.

In order to deal with the complexity of managing objects, people developed a new class of systems called ORM. Our previous example can be expressed as an ORM system with a Person class, a Address class and a PhoneNumber class, where each class maps to a table in the underlying database. Instead of writing tedious database interfacing code yourself, an ORM takes care of these issues for you while you can focus on programming the logics of the system.

## The Old Way of Writing Database Code in Python

We're going to use the library sqlite3 to create a simple database with two tables Person and Address in the following design:



Note: If you want to checkout how to use SQLite for Python, you might want to have a look at the [SQLite in Python](https://www.pythoncentral.io/introduction-to-sqlite-in-python/) series.

In this design, we have two tables person and address and address.person\_id is a foreign key to the persontable. Now we write the corresponding database initialization code in a file sqlite\_ex.py.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import sqlite3  conn = sqlite3.connect('example.db')    c = conn.cursor()  c.execute('''            CREATE TABLE person            (id INTEGER PRIMARY KEY ASC, name varchar(250) NOT NULL)            ''')  c.execute('''            CREATE TABLE address            (id INTEGER PRIMARY KEY ASC, street\_name varchar(250), street\_number varchar(250),             post\_code varchar(250) NOT NULL, person\_id INTEGER NOT NULL,             FOREIGN KEY(person\_id) REFERENCES person(id))            ''')    c.execute('''            INSERT INTO person VALUES(1, 'pythoncentral')            ''')  c.execute('''            INSERT INTO address VALUES(1, 'python road', '1', '00000', 1)            ''')    conn.commit()  conn.close() |

Notice that we have inserted one record into each table. Run the following command in your shell.

|  |  |
| --- | --- |
| 1 | $ python sqlite\_ex.py |

Now we can query the database example.db to fetch the records. Write the following code in a file sqlite\_q.py.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import sqlite3  conn = sqlite3.connect('example.db')    c = conn.cursor()  c.execute('SELECT \* FROM person')  print c.fetchall()  c.execute('SELECT \* FROM address')  print c.fetchall()  conn.close() |

And run the following statement in your shell.

|  |  |
| --- | --- |
| 1  2  3 | $ python sqlite\_q.py  [(1, u'pythoncentral')]  [(1, u'python road', u'1', u'00000', 1)] |

In the previous example, we used an sqlite3 connection to commit the changes to the database and a sqlite3 cursor to execute raw SQL statements to CRUD (create, read, update and delete) data in the database. Although the raw SQL certainly gets the job done, it is not easy to maintain these statements. In the next section, we're going to use SQLAlchemy's declarative to map the Person and Address tables into Python classes.

## Python's SQLAlchemy and Declarative

There are three most important components in writing SQLAlchemy code:

* A Table that represents a table in a database.
* A mapper that maps a Python class to a table in a database.
* A class object that defines how a database record maps to a normal Python object.

Instead of having to write code for Table, mapper and the class object at different places, SQLAlchemy's declarative allows a Table, a mapper and a class object to be defined at once in one class definition.

The following declarative definitions specify the same tables defined in sqlite\_ex.py:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import os  import sys  from sqlalchemy import Column, ForeignKey, Integer, String  from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy.orm import relationship  from sqlalchemy import create\_engine    Base = declarative\_base()    class Person(Base):      \_\_tablename\_\_ = 'person'      # Here we define columns for the table person      # Notice that each column is also a normal Python instance attribute.      id = Column(Integer, primary\_key=True)      name = Column(String(250), nullable=False)    class Address(Base):      \_\_tablename\_\_ = 'address'      # Here we define columns for the table address.      # Notice that each column is also a normal Python instance attribute.      id = Column(Integer, primary\_key=True)      street\_name = Column(String(250))      street\_number = Column(String(250))      post\_code = Column(String(250), nullable=False)      person\_id = Column(Integer, ForeignKey('person.id'))      person = relationship(Person)    # Create an engine that stores data in the local directory's  # sqlalchemy\_example.db file.  engine = create\_engine('sqlite:///sqlalchemy\_example.db')    # Create all tables in the engine. This is equivalent to "Create Table"  # statements in raw SQL.  Base.metadata.create\_all(engine) |

Save the previous code into a file sqlalchemy\_declarative.py and run the following command in your shell:

|  |  |
| --- | --- |
| 1 | $ python sqlalchemy\_declarative.py |

Now a new sqlite3 db file called "sqlalchemy\_example.db" should be created in your current directory. Since the sqlalchemy db is empty right now, let's write some code to insert records into the database:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker    from sqlalchemy\_declarative import Address, Base, Person    engine = create\_engine('sqlite:///sqlalchemy\_example.db')  # Bind the engine to the metadata of the Base class so that the  # declaratives can be accessed through a DBSession instance  Base.metadata.bind = engine    DBSession = sessionmaker(bind=engine)  # A DBSession() instance establishes all conversations with the database  # and represents a "staging zone" for all the objects loaded into the  # database session object. Any change made against the objects in the  # session won't be persisted into the database until you call  # session.commit(). If you're not happy about the changes, you can  # revert all of them back to the last commit by calling  # session.rollback()  session = DBSession()    # Insert a Person in the person table  new\_person = Person(name='new person')  session.add(new\_person)  session.commit()    # Insert an Address in the address table  new\_address = Address(post\_code='00000', person=new\_person)  session.add(new\_address)  session.commit() |

Save the previous code into a local file sqlalchemy\_insert.py and run the command python sqlalchemy\_insert.py in your shell. Now we have one Person object and one Address object stored in the database. Let's query the database using the classes defined in sqlalchemy\_declarative.py:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | >>> from sqlalchemy\_declarative import Person, Base, Address  >>> from sqlalchemy import create\_engine  >>> engine = create\_engine('sqlite:///sqlalchemy\_example.db')  >>> Base.metadata.bind = engine  >>> from sqlalchemy.orm import sessionmaker  >>> DBSession = sessionmaker()  >>> DBSession.bind = engine  >>> session = DBSession()  >>> # Make a query to find all Persons in the database  >>> session.query(Person).all()  [<sqlalchemy\_declarative.Person object at 0x2ee3a10>]  >>>  >>> # Return the first Person from all Persons in the database  >>> person = session.query(Person).first()  >>> person.name  u'new person'  >>>  >>> # Find all Address whose person field is pointing to the person object  >>> session.query(Address).filter(Address.person == person).all()  [<sqlalchemy\_declarative.Address object at 0x2ee3cd0>]  >>>  >>> # Retrieve one Address whose person field is point to the person object  >>> session.query(Address).filter(Address.person == person).one()  <sqlalchemy\_declarative.Address object at 0x2ee3cd0>  >>> address = session.query(Address).filter(Address.person == person).one()  >>> address.post\_code  u'00000' |

## Summary of Python's SQLAlchemy

In this article, we learned how to write database code using SQLAlchemy's declaratives. Compared to writing the traditional raw SQL statements using sqlite3, SQLAlchemy's code is more object-oriented and easier to read and maintain. In addition, we can easily create, read, update and delete SQLAlchemy objects like they're normal Python objects.

You might be wondering that if SQLAlchemy's just a thin layer of abstraction above the raw SQL statements, then it's not very impressive and you might prefer to writing raw SQL statements instead. In the following articles of this series, we're going to investigate various aspects of SQLAlchemy and compare it against raw SQL statements when they're both used to implement the same functionalities. I believe at the end of this series, you will be convinced that SQLAlchemy is superior to writing raw SQL statements.