How to Create a Database

Creating a database with SQLAlchemy is really easy. SQLAlchemy uses a **Declarative** method for creating databases. We will write some code to generate the database and then explain how the code works. If you want a way to view your SQLite database, I would recommend the SQLite Manager plugin for Firefox. Here's some code to create our database tables:

```
# table_def.py
from sqlalchemy import create_engine, ForeignKey
from sqlalchemy import Column, Date, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, backref

engine = create_engine('sqlite:///mymusic.db', echo=True)
Base = declarative_base()

class Artist(Base):
    """"""
    __tablename__ = "artists"

    id = Column(Integer, primary_key=True)
    name = Column(String)

class Album(Base):
    """"""
    __tablename__ = "albums"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    release_date = Column(Date)
    publisher = Column(String)
    media_type = Column(String)

    artist_id = Column(Integer, ForeignKey("artists.id"))
    artist = relationship("Artist", backref=backref("albums", order_by=id))

# create tables
Base.metadata.create_all(engine)
```

If you run this code, then you should see something similar to the following output:

```
2014-04-03 09:43:57,541 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' A
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns'
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("artists")
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("albums")
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine
CREATE TABLE artists (
    id INTEGER NOT NULL,
    name VARCHAR,
    PRIMARY KEY (id)
)


2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
2014-04-03 09:43:57,661 INFO sqlalchemy.engine.base.Engine COMMIT
2014-04-03 09:43:57,661 INFO sqlalchemy.engine.base.Engine
CREATE TABLE albums (
    id INTEGER NOT NULL,
    title VARCHAR,
    release_date DATE,
    publisher VARCHAR,
    media_type VARCHAR,
    artist_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY(artist_id) REFERENCES artists (id)
)


2014-04-03 09:43:57,661 INFO sqlalchemy.engine.base.Engine ()
2014-04-03 09:43:57,741 INFO sqlalchemy.engine.base.Engine COMMIT
```

Why did this happen? Because when we created the engine object, we set its **echo** parameter to **True**. The engine is where the database connection information is and it has all the DBAPI stuff in it that makes communication with your database possible. You'll note that we're creating a SQLite database. Ever since Python 2.5, SQLite has been supported by the language. If you want to connect to some other database, then you'll need to edit the connection string. Just in case you're confused about what we're talking about, here is the code in question:

```
engine = create_engine('sqlite:///mymusic.db', echo=True)
```

The string, **sqlite:///mymusic.db**, is our connection string. Next we create an instance of the declarative base, which is what we'll be basing our table classes on. Next we have two classes, **Artist** and **Album** that define what our database tables will look like. You'll notice that we have **Columns**, but no

column names. SQLAlchemy actually used the variable names as the column names unless you specifically specify one in the **Column** definition. You'll note that we are using an **id** Integer field as our primary key in both classes. This field will auto-increment. The other columns are pretty self-explanatory until you get to the **ForeignKey**. Here you'll see that we're tying the **artist_id** to the **id** in the **Artist** table. The relationship directive tells SQLAlchemy to tie the Album class/table to the Artist table. Due to the way we set up the ForeignKey, the relationship directive tells SQLAlchemy that this is a **many-to-one relationship**, which is what we want. Many albums to one artist. You can read more about table relationships here.

The last line of the script will create the tables in the database. If you run this script multiple times, it won't do anything new after the first time as the tables are already created. You could add another table though and then it would create the new one.