Fairfield University
DOLAN
School of Business

jupyter

NUMFOCUS
OPEN CODE = BETTER SCIENCE

SQLAlchemy

Databases for Analytics

SqlAlchemy and pandas tips
for Pythonistas

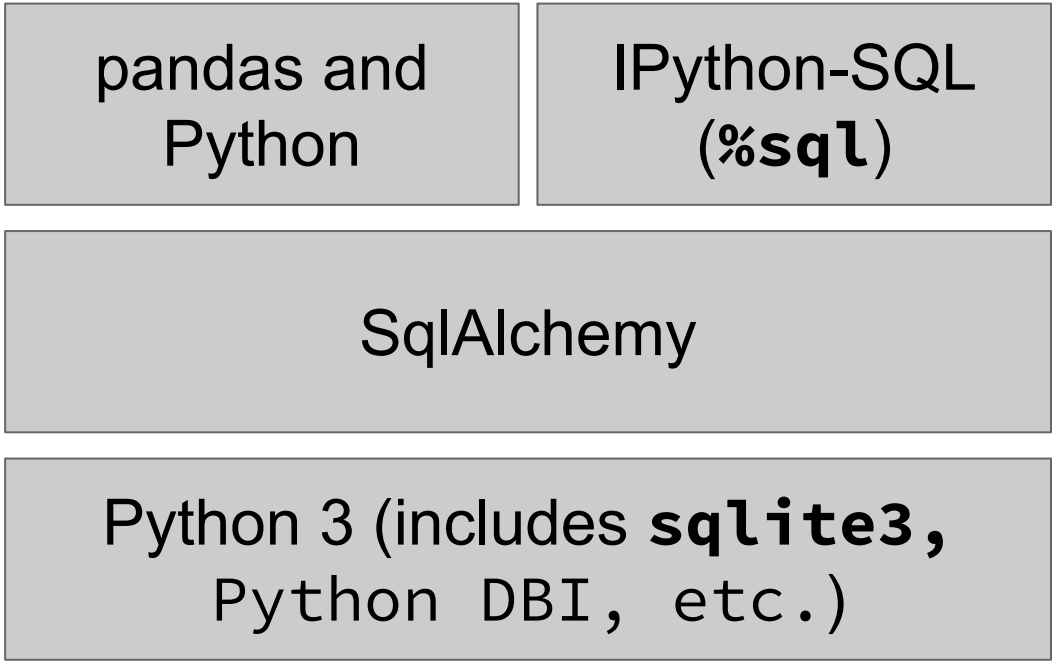
Learning Objectives

- **Skills:** You should know how to ...
  - Configure and use SqlAlchemy
  - Use pandas to glue relational and non-relational data
  - Extract, Transform, Load data with pandas or SqlAlchemy
- **Theory:** You should be able to explain ...
  - How the layers of the Python → SqlAlchemy → pandas stack work together to simplify data ETL
  - Basic functions of an Object-Relational Mapper

Stacked Architecture

Each layer is built on top of the ones below it, providing greater convenience and ease of use …

But you can always go deeper if needed.



SqlAlchemy

The de facto standard for RDBMS Access in Python

SqlAlchemy

SQLAlchemy is a library for accessing SQL databases in Python.

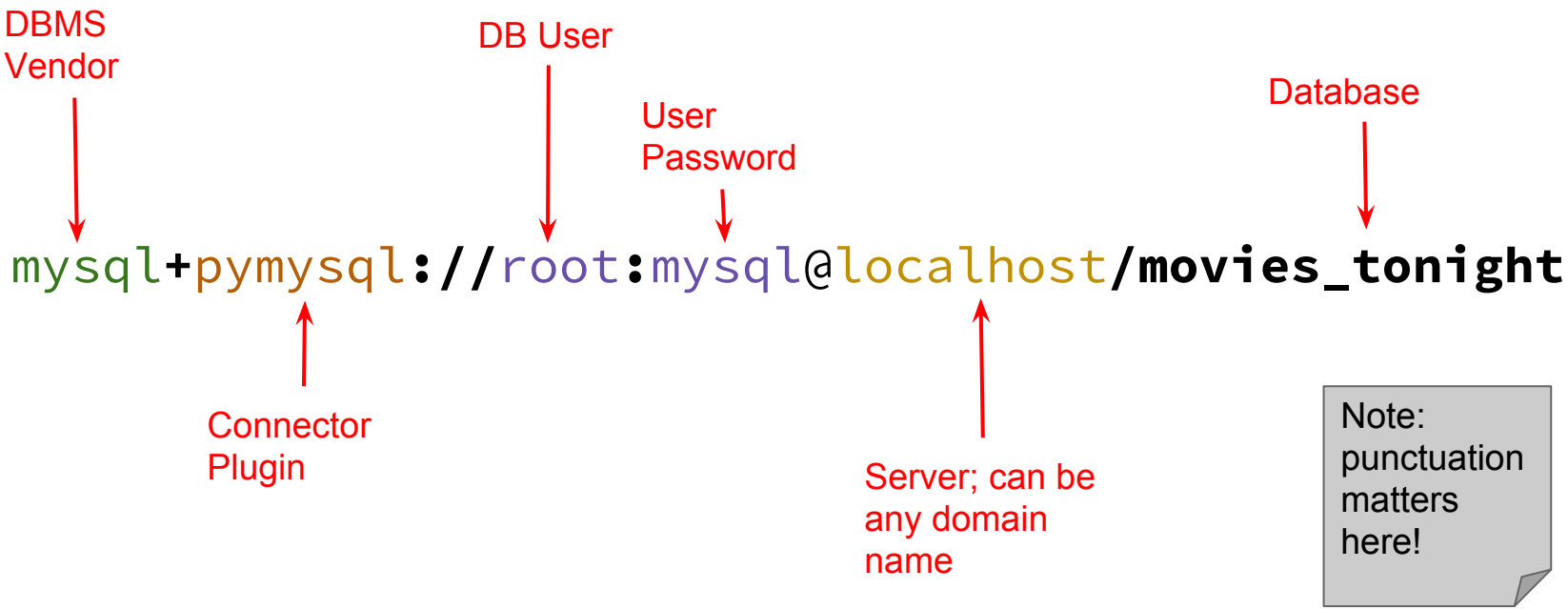
- Provides a plug-in system for connecting to various DBMSes.
- Works as an Object Relational Mapper for converting Python → SQL and SQL→ Python
- Key is creating a SQLAlchemy Engine for creating and configuring DB connections as needed.

Engines

An engine is a reusable configuration for connecting to a database. You create one before doing anything else.

```
from sqlalchemy import create_engine
engine = create_engine("mysql+pymysql://root:mysql@localhost/movies_tonight")
# We can now use engine to create new database connections
```

Connection Strings



Sessions

Each connection to a database is a *session*.

One can open multiple sessions at the same time.

Each database operation (DDL or DML) exists within a session context.

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session() # session is a connection
```

ORM: Python → SQL DDL

SQLAlchemy provides full support for most SQL DDL statements. You can even use it to create SQL tables based on a Python class:

```
from sqlalchemy import Column, Integer, String
class Artist(Base):
    __tablename__ = 'artists'
    id = Column(Integer, primary_key=True)
    name = Column(String)
Base.metadata.create_all(engine) # Writes SQL DDL
```

ORM: Python → SQL DML

SQLAlchemy also provides Python equivalents to SQL **`INSERT`**, **`UPDATE`**, and **`DELETE`** commands.

The idea is that you create/update/delete Python ***objects*** and then ***commit*** changes to the database.

```
artists_list = [Artist(name='Minnie Driver'),
                Artist(name='STanley Tucci'),...]
session.add.all(artists_list)
session.commit()
```

Artist class was defined on previous slide.

ORM: SQL Data → Python Objects

SQLAlchemy sessions have a **`query()`** method for retrieving data as lists of Python objects.

```
# query() returns a list of objects of the specified type
artist_list = session.query(Artist).order_by(Artist.name)
```

Raw SQL Queries

One can also get away without the ORM by executing SQL directly, which is usually what you want anyway. The result is a list of ***tuples***.

Triple-quoted query string with line breaks

```
from sqlalchemy.sql import text
s = text('''SELECT `Name`
            FROM `ARTISTS`''')
conn.execute(s).fetchall()
   → [(u'Minnie Driver'),(u'Stanley Tucci'), … ]
```

Expanded Data Types for SQLite

SqlAlchemy expands the list of supported data types:

```
BLOB, BOOLEAN, CHAR, DATE, DATETIME, DECIMAL,
FLOAT, INTEGER, NULL, NUMERIC, SMALLINT,
TEXT, TIME, TIMESTAMP, VARCHAR
```

In some cases it uses SQLite's own type conversion functions. In others (`DATE, DATETIME,TIME`) SqlAlchemy uses its own custom conversion functions.

All storage is still in the five native types, however.

More about SQLAlchemy

SQLAlchemy has lots and lots more tricks …

You can RTFM or just follow along with the tutorial:

http://docs.sqlalchemy.org/en/latest/orm/tutorial.html

Pandas

Useful SQL and RDBMS-like features

Pandas & SQLAlchemy

Pandas uses the SQLAlchemy library as the basis for for its `read_sql()`, `read_sql_table()`, `and read_sql_query()` functions.

- This allows you to retrieve query results as a Pandas DataFrame
- However, you need to initiate the database connection with SQLAlchemy first

`read_sql()` Example

pandas also supports the native **`sqlite3`** connector

```
import pandas as pd
from sqlalchemy import create_engine
engine = create_engine("mysql+pymysql://root:mysql@localhost/deals")

with engine.connect() as conn, conn.begin():
    deals = pd.read_sql('Select * from Deals', conn)
    players=pd.read_sql('Select * from Players', conn)
    companies=pd.read_sql('Select * from Companies', conn)
# deals, players, and companies are Pandas DataFrames
```

merge() and .join()

Pandas supports its own **join** syntax for DataFrames.

```
# using the Pandas merge() function, one join at a time
deal_players = pd.merge(deals,players,on='DealID')
deal_companies =
    pd.merge(deal_players,companies,on='CompanyID')

# using the DataFrame join() method, which can be chained
deal_players2 =
    deals.join(players,on='DealID').join(companies,on='CompanyID')
```

Mixing SQL Data with DataFrames

Pandas `merge()` and `join()` work like `SELECT` queries with `JOINS`, except with `DataFrames` instead of tables.

So, who says that the `DataFrames` have to come from a SQL query? They can come from anywhere!

So, we can now glue data together from multiple sources, even if some of them are non-relational.

Writing Data with `.to_sql()`

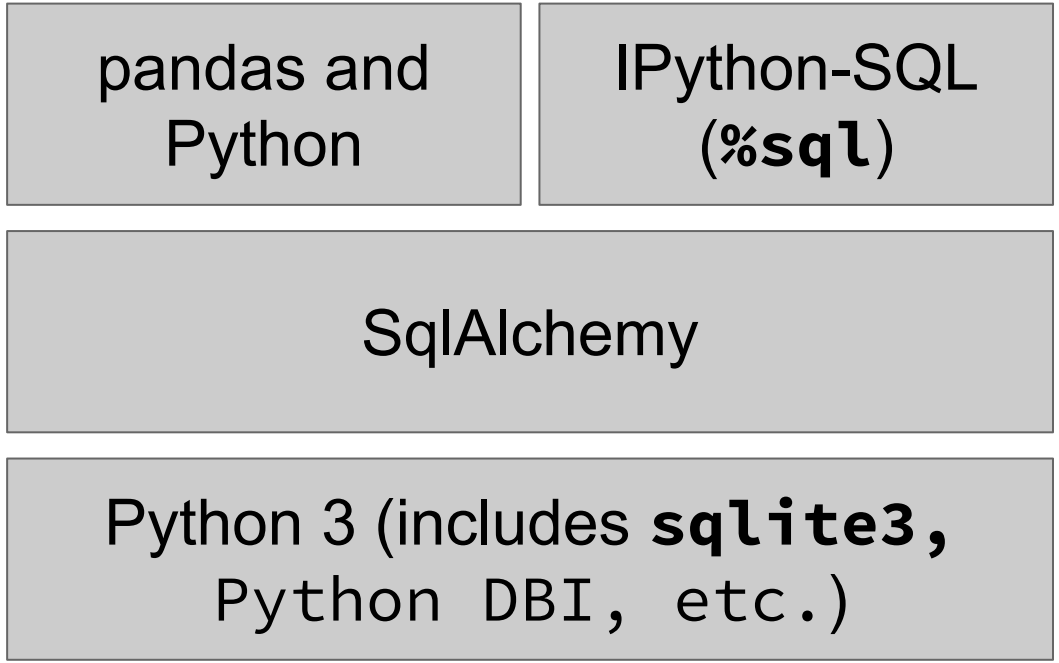
pandas can write directly to the database (using a special SqlAlchemy wrapper).

For example ...

```
conn = sqlite3.connection('MyDatabase.db')
my_table = pd.read_csv('my_data.csv')
# … do some data cleanup & munging …
my_table.to_sql('MY_TABLE',conn,
                if_exists='append',index=false)
```

Stacked Architecture (again)

If you are skilled in all three layers, many ETL tasks reduce to a few lines of code.



Fairfield University
DOLAN
School of Business

jupyter

NUMFOCUS
OPEN CODE = BETTER SCIENCE

SQLAlchemy

Databases for Analytics

SqlAlchemy and pandas tips
for Pythonistas