



Introduction to SQLAlchemy and ORMs

Mike Bayer

@zzzeek

Pycon US 2013

Updates!

- Break is at 2:50 PM
- On-Site Volunteering opportunities!
 - SWAG bagging - Thursday, 4-8 PM
 - Registration Desk help
 - <http://bit.ly/pycon-volunteering-status>
 - <http://bit.ly/pycon2013-volunteer>

Prerequisites

```
git clone https://bitbucket.org/zzzeek/  
pycon2013_student_package.git
```

```
cd pycon2013_student_package/
```

```
virtualenv --no-site-packages .venv
```

```
.venv/bin/python install.py
```

```
.venv/bin/sliderepl 01_engine_usage.py
```

Relational Database Review

- Any questions on the handout?
- We've covered:
 - Tables, columns
 - data definition language (DDL)
 - INSERT, UPDATE, DELETE (DML)
 - SELECT, joins, grouping
 - Overview of ACID

SQLAlchemy – Overview

- the Database Toolkit for Python
- introduced 2005
- end-to-end system for working with the Python DBAPI, relational databases, and the SQL language
- Current release 0.8.0
- 1.0 will happen

SQLAlchemy Goals

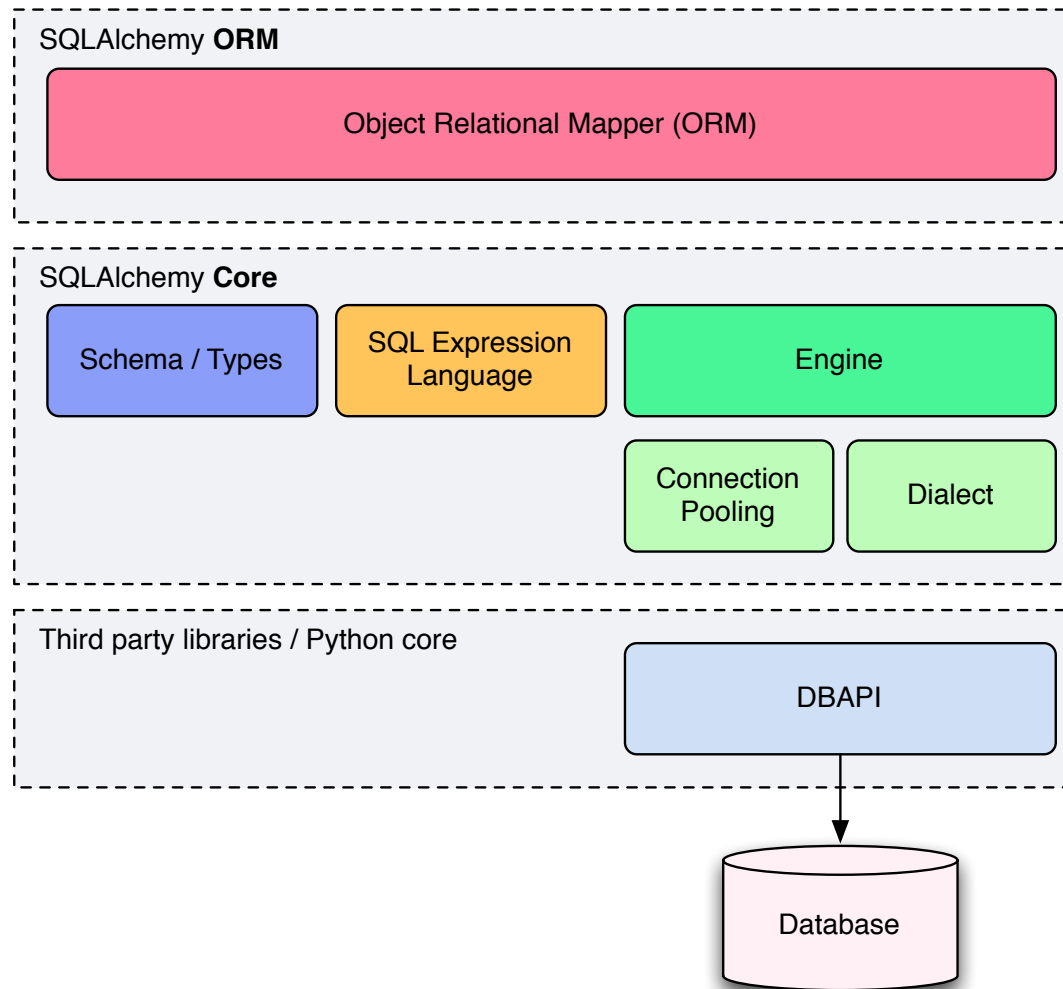
- Provide helpers, tools and components to assist with database development at every level
- Provide a consistent and fully featured facade over the Python DBAPI
- Provide an industrial strength, but optional, object relational mapper (ORM)
- Act as the foundation for any number of third party or in-house tools

SQLAlchemy Philosophies

- Bring the usage of different databases and adapters to an interface as consistent as possible...
- ...but still expose distinct behaviors and features of each backend.
- Never "hide" the database or its concepts - developers must know / continue to think in SQL...
- Instead....provide automation and DRY
- Allow expression of DB/SQL tasks using declarative patterns

SQLAlchemy Overview

SQLAlchemy consists of the **Core** and the **ORM**



SQLAlchemy – Core

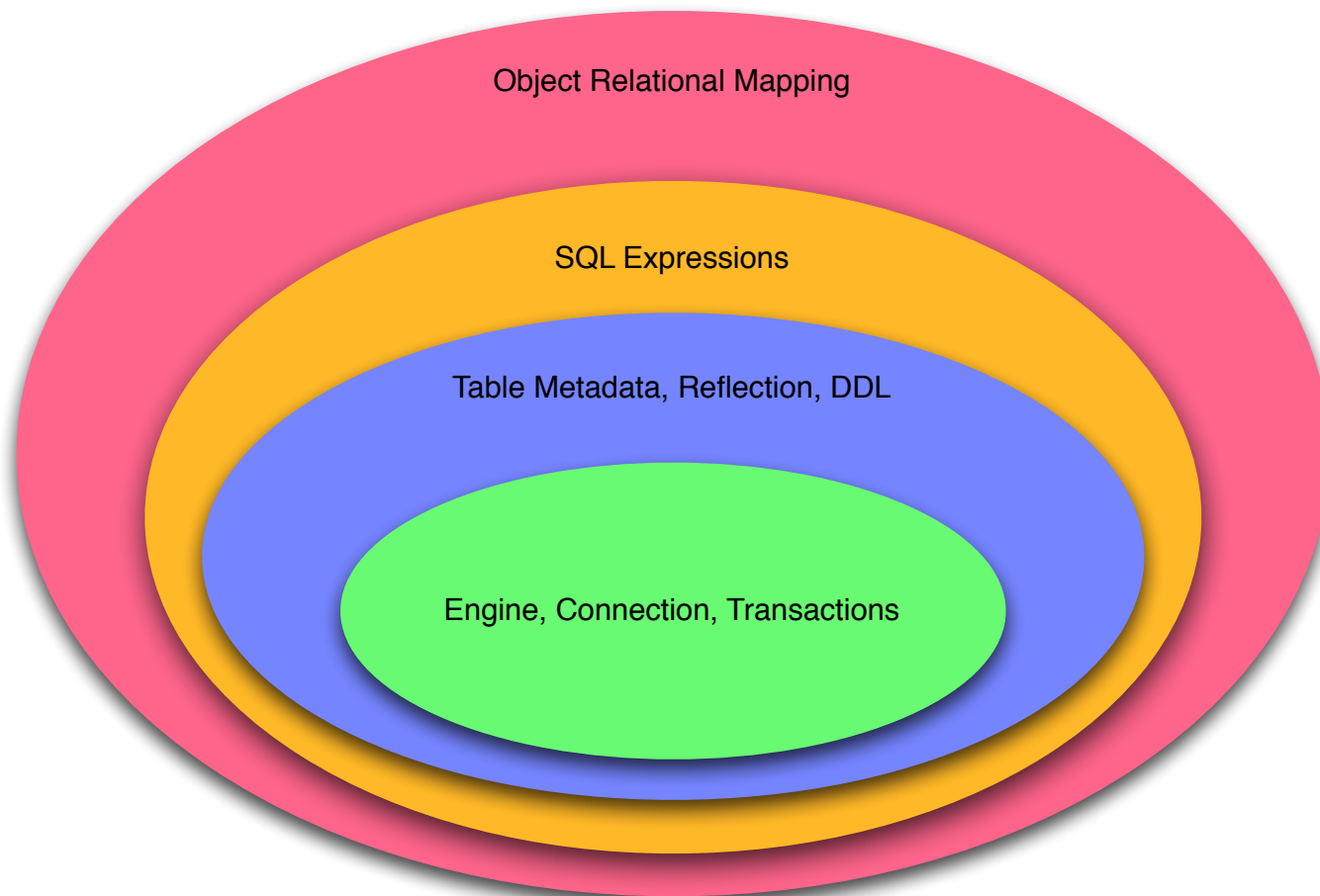
- **Engine** - a registry which provides connectivity to a particular database server.
- **Dialect** - interprets generic SQL and database commands in terms of a specific DBAPI and database backend.
- **Connection Pool** - holds a collection of database connections in memory for fast re-use.
- **SQL Expression Language** - Allows SQL statements to be written using Python expressions
- **Schema / Types** - Uses Python objects to represent tables, columns, and datatypes.

SQLAlchemy – ORM

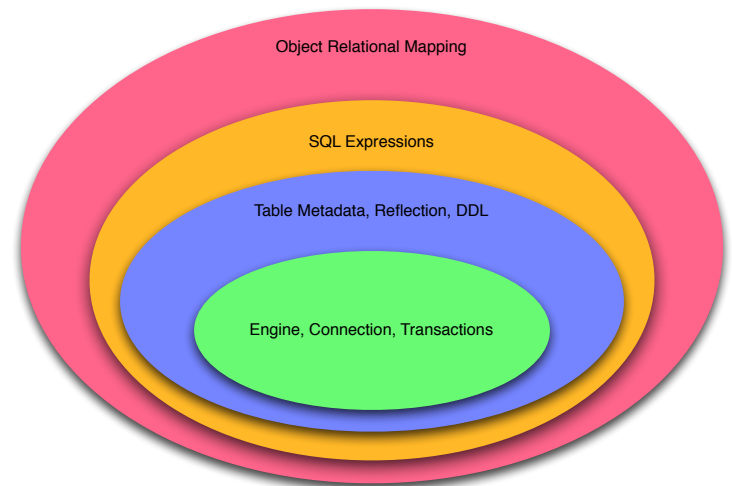
- Allows construction of Python objects which can be **mapped** to relational database tables.
- Transparently persists objects into their corresponding database tables using the **unit of work** pattern.
- Provides a query system which loads objects and attributes using SQL generated from mappings.
- Builds on top of the **Core** - uses the Core to generate SQL and talk to the database.

SQLAlchemy is like an Onion

Can be learned from the inside out, or outside in



Level 1, Engine, Connection, Transactions



The Python DBAPI

- DBAPI - PEP-0249, Python Database API
- The de-facto system for providing Python database interfaces
- There are many DBAPI implementations available, most databases have more than one
- Features/performance/stability/API quirks/maintenance vary wildly

DBAPI – Nutshell

```
import psycopg2
connection = psycopg2.connect("scott", "tiger", "test")

cursor = connection.cursor()
cursor.execute(
    "select emp_id, emp_name from employee "
    "where emp_id=%(emp_id)s",
    {'emp_id':5})
emp_name = cursor.fetchone()[1]
cursor.close()

cursor = connection.cursor()
cursor.execute(
    "insert into employee_of_month "
    "(emp_name) values (%(emp_name)s)",
    {"emp_name":emp_name})
cursor.close()

connection.commit()
```

Important DBAPI Facts

- DBAPI assumes that a *transaction is always in progress*. There is no `begin()` method, only `commit()` and `rollback()`.
- DBAPI encourages bound parameters, via the `execute()` and `executemany()` methods. But has six different formats.
- All DBAPIs have inconsistencies regarding datatypes, primary key generation, custom database features, result/cursor behavior
- DBAPI has it's own exception hierarchy, which SQLAlchemy exposes directly

SQLAlchemy and the DBAPI

- The first layer in SQLAlchemy is known as the **Engine**, which is the object that maintains the classical DBAPI interaction.

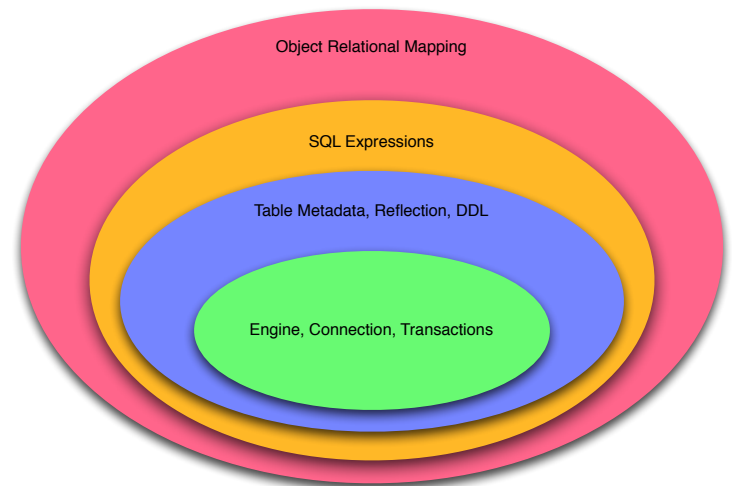
Engine – Usage

```
.venv/bin/sliderepl 01_engine.py
```

Engine Facts

- Executing via the Engine directly is called **connectionless execution** - the Engine connects and disconnects for us.
- Using a Connection is called **explicit execution**. We control the span of a connection in use.
- Engine usually uses a connection pool, which means "disconnecting" often means the connection is just returned to the pool.
- The SQL we send to engine.execute() as a string is not modified, is consumed by the DBAPI verbatim.

Level 2, Table Metadata, Reflection, DDL



What is "Metadata"?

- Popularized by Martin Fowler, *Patterns of Enterprise Architecture*
- Describes the structure of the database, i.e. tables, columns, constraints, in terms of data structures in Python
- Serves as the basis for SQL generation and object relational mapping
- Can generate *to* a schema
- Can be generated *from* a schema

MetaData and Table

```
.venv/bin/sliderepl 02_metadata.py
```

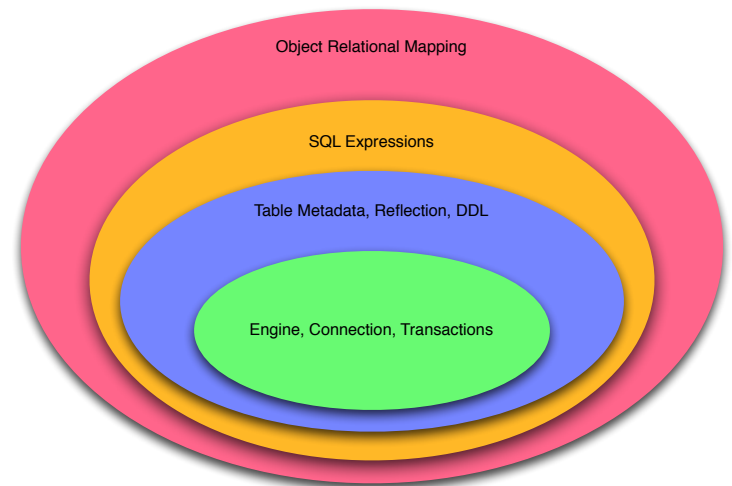
Some Basic Types

- `Integer ()` - basic integer type, generates INT
- `String ()` - ASCII strings, generates VARCHAR
- `Unicode ()` - Unicode strings - generates VARCHAR, NVARCHAR depending on database
- `Boolean()` - generates BOOLEAN, INT, TINYINT
- `DateTime ()` - generates DATETIME or TIMESTAMP, returns Python `datetime ()` objects
- `Float ()` - floating point values
- `Numeric ()` - precision numerics using Python `Decimal ()`

CREATE and DROP

- `metadata.create_all(engine, checkfirst=<True | False>)` emits CREATE statements for all tables.
- `table.create(engine, checkfirst=<True | False>)` emits CREATE for a single table.
- `metadata.drop_all(engine, checkfirst=<True | False>)` emits DROP statements for all tables.
- `table.drop(engine, checkfirst=<True | False>)` emits DROP for a single table.

Level 3, SQL Expressions



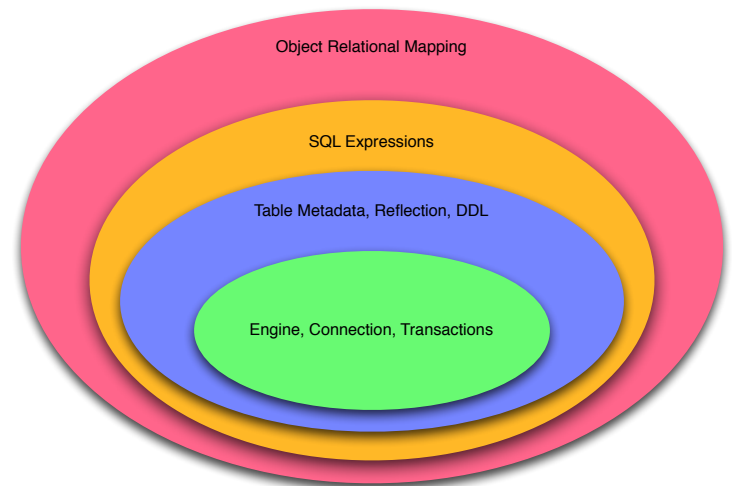
SQL Expressions

- The **SQL Expression** system builds upon **Table Metadata** in order to compose SQL statements in Python.
- We will build Python objects that represent individual SQL strings (statements) we'd send to the database.
- These objects are composed of other objects that each represent some unit of SQL, like a comparison, a SELECT statement, a conjunction such as AND or OR.
- We work with these objects in Python, which are then converted to strings when we "execute" them (as well as if we `print` them).

SQL Expressions

```
.venv/bin/sliderepl 03_sql_expressions.py
```

Level 4, Object Relational Mapping

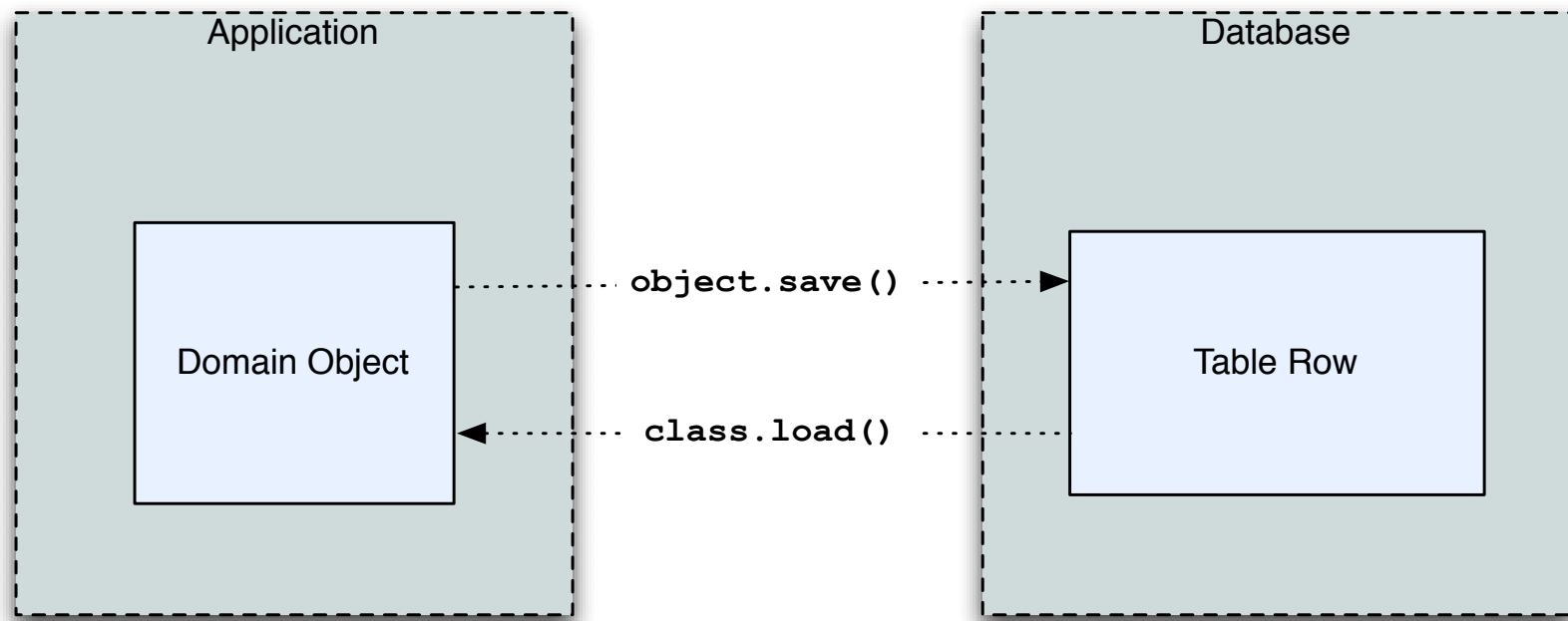


Object Relational Mapping

- Object Relational Mapping, or ORM, is the process of associating object oriented classes with database tables.
- We refer to the set of object oriented classes as a **domain model**.

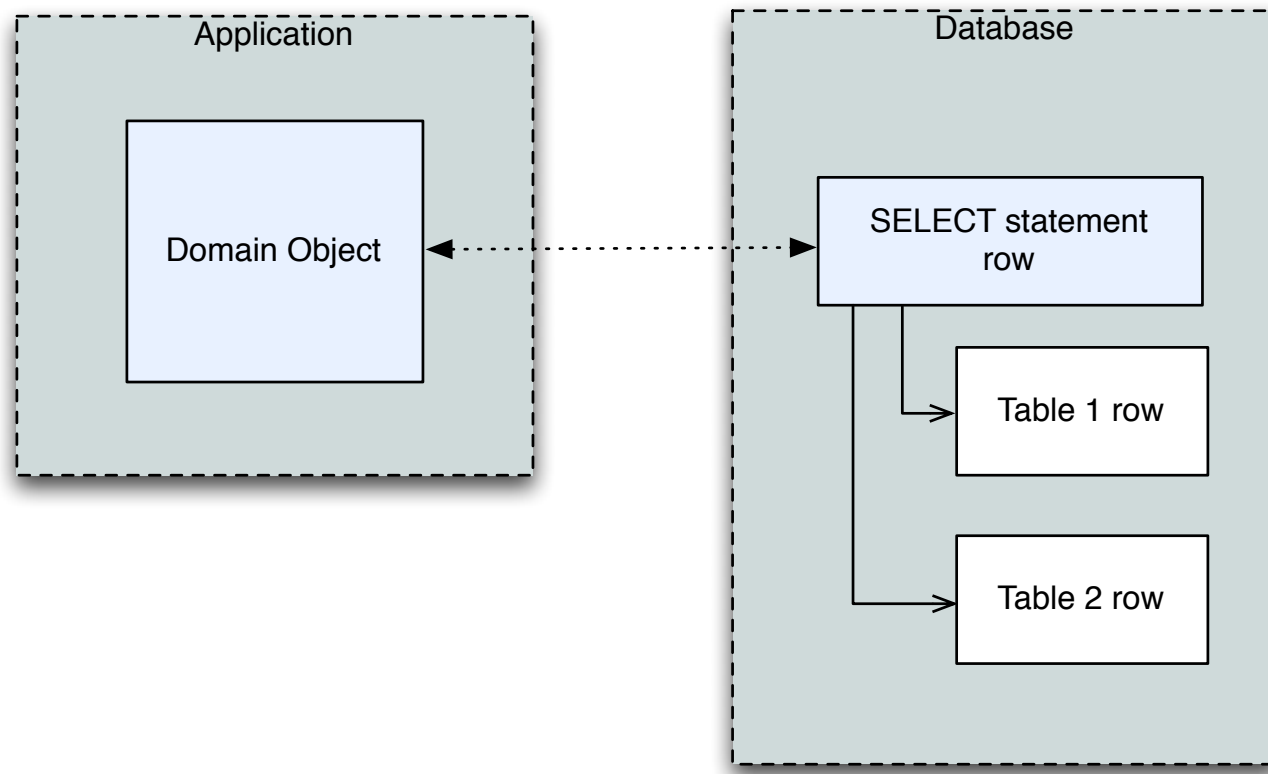
What does an ORM Do?

The most basic task is to translate between a domain object and a table row.



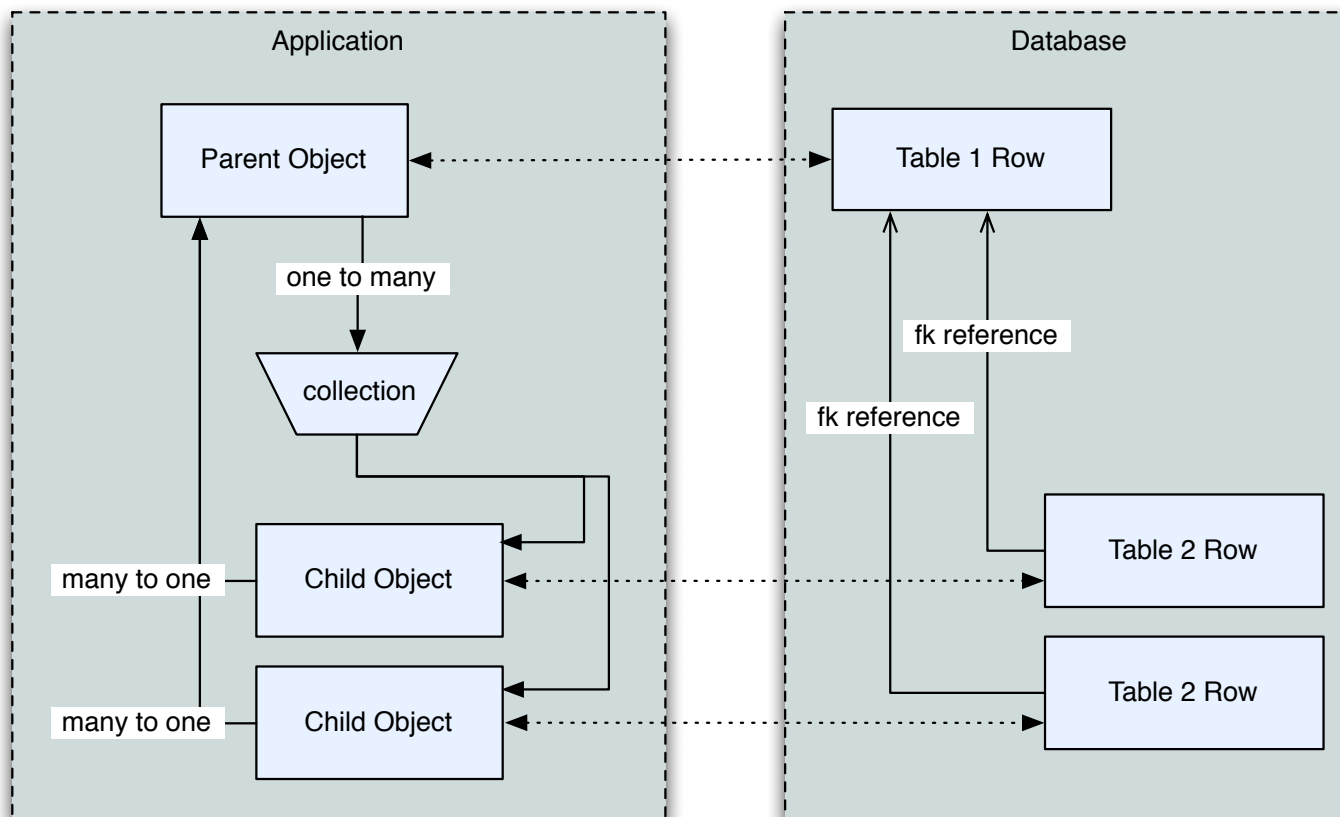
What does an ORM Do?

Some ORMs can also represent *arbitrary* rows as domain objects within the application, that is, rows derived from SELECT statements or views.



What does an ORM Do?

Most ORMs also represent basic *compositions*, primarily **one-to-many** and **many-to-one**, using **foreign key associations**.



What does an ORM Do?

- Other things ORMs do:
 - provide a means of querying the database in terms of the domain model structure
 - Some can represent class inheritance hierarchies using a variety of schemes
 - Some can handle "sharding" of data, i.e. storing a domain model across multiple schemas or databases
 - Provide various patterns for concurrency, including row versioning
 - Provide patterns for data validation and coercion

Flavors of ORM

The two general styles of ORM are **Active Record** and **Data Mapper**. **Active Record** has domain objects handle their own persistence:

```
user_record = User(name="ed", fullname="Ed Jones")
user_record.save()
```

```
user_record = User.query(name='ed').fetch()
user_record.fullname = "Edward Jones"
user_record.save()
```

Flavors of ORM

The **Data Mapper** approach tries to keep the details of persistence *separate* from the object being persisted.

```
dbsession = start_session()

user_record = User(name="ed", fullname="Ed Jones")

dbsession.add(user_record)

user_record = dbsession.query(User).filter(name='ed').first()
user_record.fullname = "Edward Jones"

dbsession.commit()
```

Flavors of ORM

ORMs may also provide different configurational patterns. Most use an "all-at-once", or **declarative** style where class and table information is together.

```
# a hypothetical declarative system
```

```
class User(ORMObject):  
    tablename = 'user'  
  
    name = String(length=50)  
    fullname = String(length=100)  
  
class Address(ORMObject):  
    tablename = 'address'  
  
    email_address = String(length=100)  
    user = many_to_one("User")
```

Flavors of ORM

A less common style keeps the declaration of domain model and table metadata separate.

```
# class is declared without any awareness of database
```

```
class User(object):  
    def __init__(self, name, username):  
        self.name = name  
        self.username = username
```

```
# elsewhere, it's associated with a database table
```

```
mapper(  
    User,  
    Table("user", metadata,  
        Column("name", String(50)),  
        Column("fullname", String(100))  
    )  
)
```

SQLAlchemy ORM

- The SQLAlchemy ORM is essentially a data mapper style ORM.
- Modern versions use declarative configuration; the "domain and schema separate" configuration model is present underneath this layer.
- The ORM builds upon SQLAlchemy Core, and many of the SQL Expression concepts are present when working with the ORM as well.
- In contrast to the SQL Expression language, which presents a schema-centric view of data, it presents a domain-model centric view of data.

Key ORM Patterns

- **Unit of Work** - objects are maintained by a system that tracks changes over the course of a transaction, and **flushes** pending changes periodically, in a transparent or semi-transparent manner
- **Identity Map** - objects are tracked by their primary key within the unit of work, and are kept *unique* on that primary key identity.
- **Lazy Loading** - Some attributes of an object may emit additional SQL queries when they are accessed.
- **Eager Loading** - Multiple tables are queried at once in order to load related objects and collections.
- **Method Chaining** - queries are composed using a string of method calls which each return a new query object.

ORM Walkthrough

```
.venv/bin/sliderepl 04_orm.py
```

SQLAlchemy



Object Relational Mapping

SQL Expressions
Thanks !

Table Metadata, Reflection, DDL

<http://www.sqlalchemy.org>

Engine, Connection, Transactions

@zzzeek