The Object Relational Mapper (ORM)

Demitri Muna
Ohio State University

12 June 2015

(Really!) Brief Intro to Object-Oriented Design

Imagine you want to represent a sphere in code. This is what it might look like using functional programming:

```
sphere = {x:0, y:0, radius:5.0}
a = area_of_sphere(sphere)
v = volume_of_sphere(sphere)

v = volume_of_cube(cube)
```

- *"sphere" is just a collection of numbers*
- *functions need to be defined separately to operate on "sphere" objects*
- *code is not organized with respect to the object*

With object oriented code, we want to express ideas more closely to our mental picture of what we're coding, rather than a list of numbers.

(Really!) Brief Intro to Object-Oriented Design

Imagine code that looks like this:

```
sphere = Sphere()
sphere.radius = 5.0
sphere.origin = Point(x=0, y=0)
a = sphere.area
v = sphere.volume

cube = Cube()
a = cube.area
v = cube.volume
```

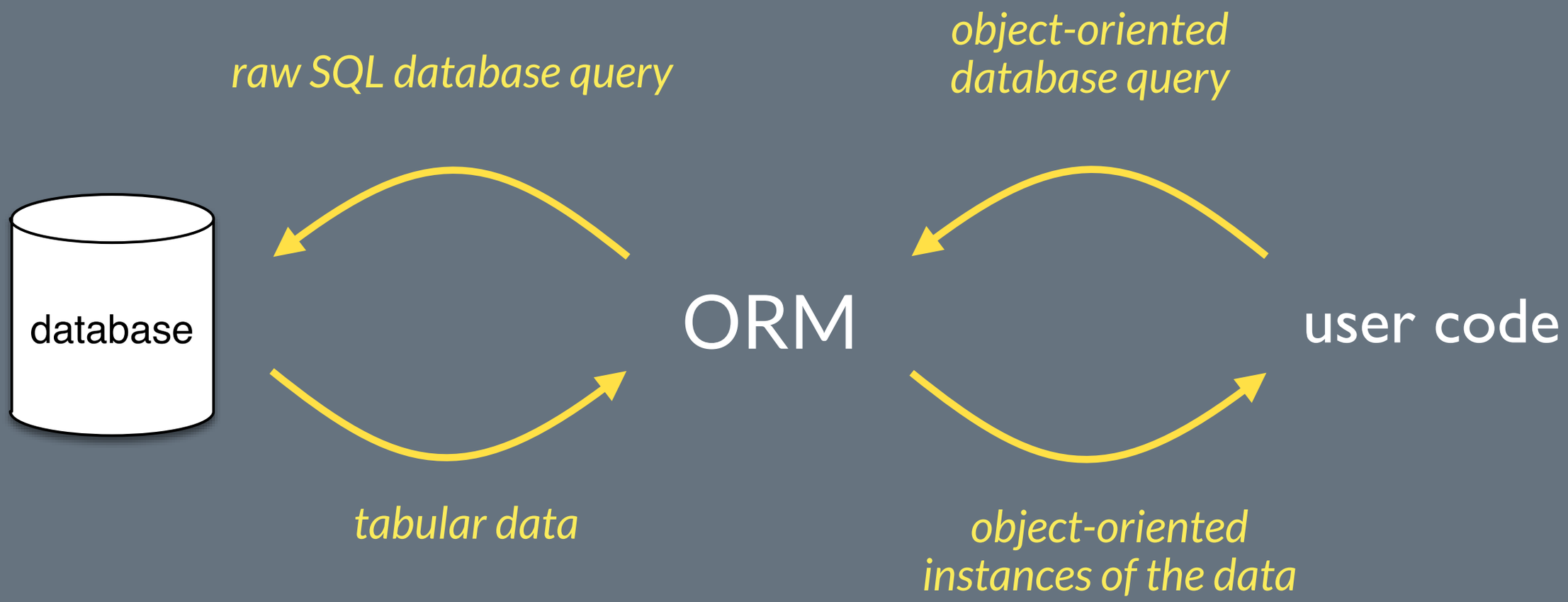
a class

- *a sphere is an opaque object; the code that defines it is responsible for its representation, not the user*
- *the sphere "knows how to do things", i.e. you don't need separate functions*
- *similar classes can have the same property names (subclasses)*
- *code is easy to read, easy to use, and more closely matches our mental model*

This is useful when working with data. I don't want to manage arrays of student names or how to store all their properties. However, the database can *only* returns tabular data (i.e. like a spreadsheet).

Object Relational Mapper (ORM)

An ORM sits between a database and objects (classes), translating high-level database queries from objects into low-level SQL code, and vice versa.



Further, these objects know about their relationships to other objects (i.e., tables in the database).

SQLAlchemy

The idea of an object-relational mapper isn't new, and there are many implementations of this idea (though none as good as the first one…). From here on we'll be talking about SQLAlchemy, a Python implementation.

Terminology Detour

session – Our interface to the database. It has a connection to the database, contains information about all of the tables, columns, data types, and relationships (*metadata*).

transaction – One or more grouped commands sent to the database. The results of these commands (e.g. adding data, deleting data) are not made permanent until a *commit* statement. A *rollback* command will undo all changes since the start of the transaction. Transactions have a "begin", but sessions can be set so that this is automatic (doesn't need to be explicitly begun).

The Session

The database is here, but we no longer care what happens on this side.

All interactions with the database are now through the Session. Data returned is in the form of objects that are tied to the Session.

Adding Data to the Database

We use the Session object as our interface to the database. Objects placed into the Session are written to the database when we "commit" the session.

```
new_student = Student()
new_student.first_name = "Gently"
new_student.last_name = "Benevolent"
session.add(new_student)

session.commit()
```

We can treat an object as being in the database after it is added to the session while still before the commit, i.e. queries that match the object will still find it.

Deleting objects from the database is very similar:

```
session.delete(some_student)
session.commit()
```

- *only takes a single object – can't delete a list of objects*
- *still must "commit()" before making permanent*

SQLAlchemy Queries

Get all rows from the "student" table, returned as a Python list of objects of type "Student".

```
students = session.query(Student).all()
```

(we will define the schema classes shortly)

```
session
.query(Student)
.all()
```

the Session class is provided by SQLAlchemy

specify in "query" object we want to get back

return all of the rows in the table in a Python list

Each item in this list knows everything about the "student" table:

```
for student in students:
    print(student.first_name)
    print(student.last_name)
    print(student.supervisors)
```

It knows about the relationship to supervisor! This is from the metadata. The result here is a list of objects of type "Supervisor".

SQLAlchemy Queries

Get all students whose first name is “Phillip”.

```
students = session.query(Student)\
               .filter(Student.first_name="Phillip").all()
```

The properties (like ‘first_name’, ‘last_name’) come from the schema and must be spelled the same way as they are defined in the database.

Filter statements can be chained:

```
students = session.query(Student)\
               .filter(Student.first_name="Phillip")\
               .filter(Student.last_name="Bin").all()
```

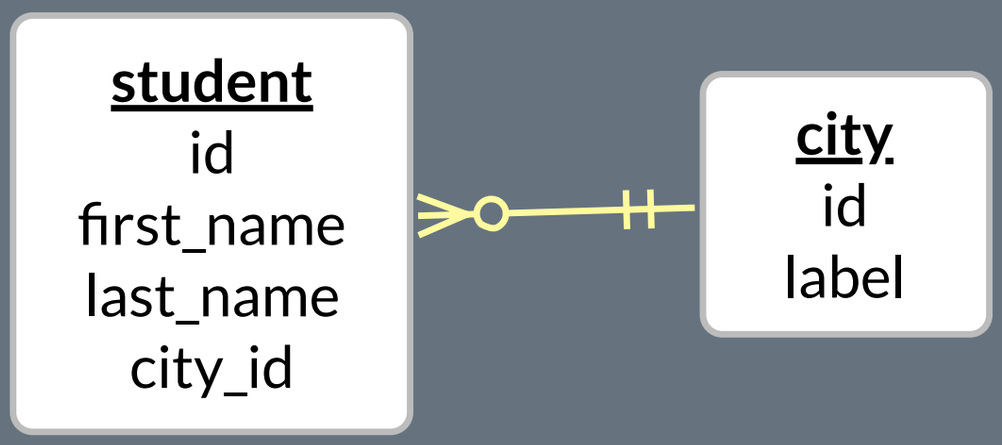
These commands are converted into raw SQL statements and sent to the database.

SQLAlchemy Queries

Tables can be connected (joined); get all students whose first name are Nedward and are in New York:

```
students = session.query(Student)\
                .join(City)\
                .filter(Student.first_name="Phillip")\
                .filter(City.label="New York")\
                .all()
```

For every line in the schema connecting tables, the table name must be specified in the "join" statement.



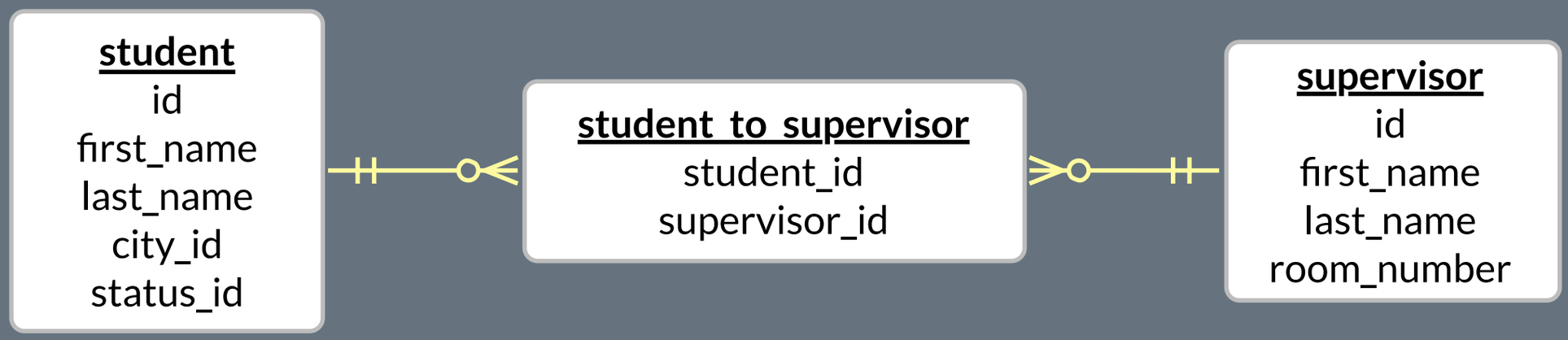
Note that we never *use the primary keys or foreign keys in queries! This is bookkeeping information, not data.*

SQLAlchemy Queries

Querying and filtering across several tables.

note the join table must be specified

```
students = session.query(Student)\
               .join(StudentToSupervisor, Supervisor)\
               .filter(Student.first_name="Clara")\
               .filter(Supervisor.last_name="Who")\
               .all()
```



Lookup Table Queries

Typically, lookup tables do not have duplicate entries (think "club"– the same club shouldn't appear twice in the table). This is how we handle this in code:

```
try:
    curling_club = session.query(Club)\
            .filter(Club.name="Curling").one()
except sqlalchemy.orm.exc.NoResultFound:
    # not already in the database, add
    curling_club = Club()
    curling_club.name = "Curling"
    session.add(curling_club)
except sqlalchemy.orm.exc.MultipleResultsFound:
    raise Exception("Hey! There are duplicate
                    Club entries! Please fix!")
```

"one()" says that we are expecting ONE AND ONLY ONE result to be found. Multiple results or no results is an error– this is appropriate for a lookup table (and is a good safety net).

At this point, 'curling_club' is defined; either newly added or fetched from the database. This code can be safely run several times.

Traversing the Schema

If you have an object from the database, you can follow any relationship (lines in the schema) to any other object *without a new query*.

```
student = session.query(Student).first()
```

Get the list of clubs a student is in.

```
student.clubs
```

the relationship is a property

Get all of the students of the first supervisor of this student.

```
student.supervisors[0].students
```

List all cities that students from the curling club are from:

```
curling_club = session.query(Club).\
                    filter(Club.name="Curling").one()
for student in curling_club.students:
    print(student.city.name)
```

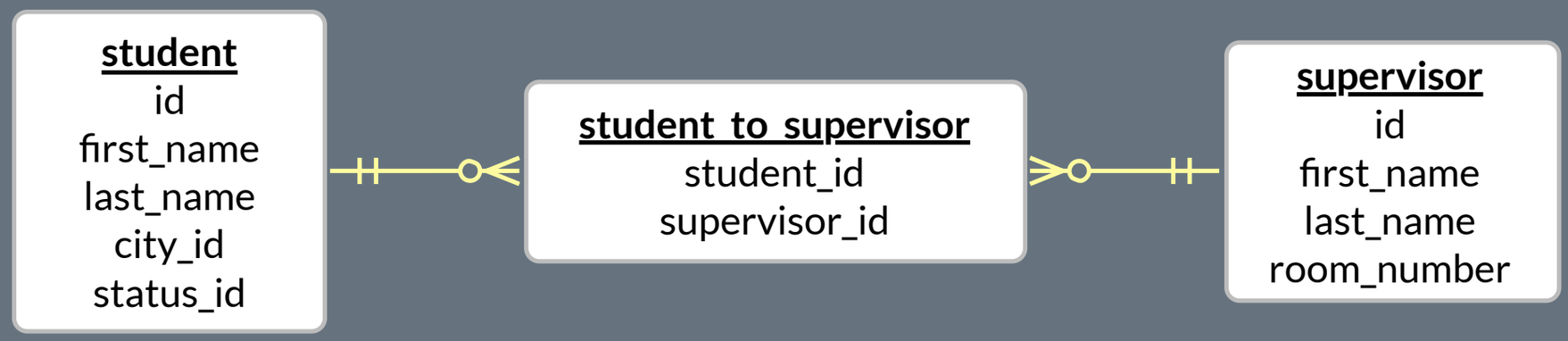
Associating Related Objects

Associating objects is as simple as you'd expect. The to-many related property is just a Python list, so to associate two objects across a relation just add it to the list:

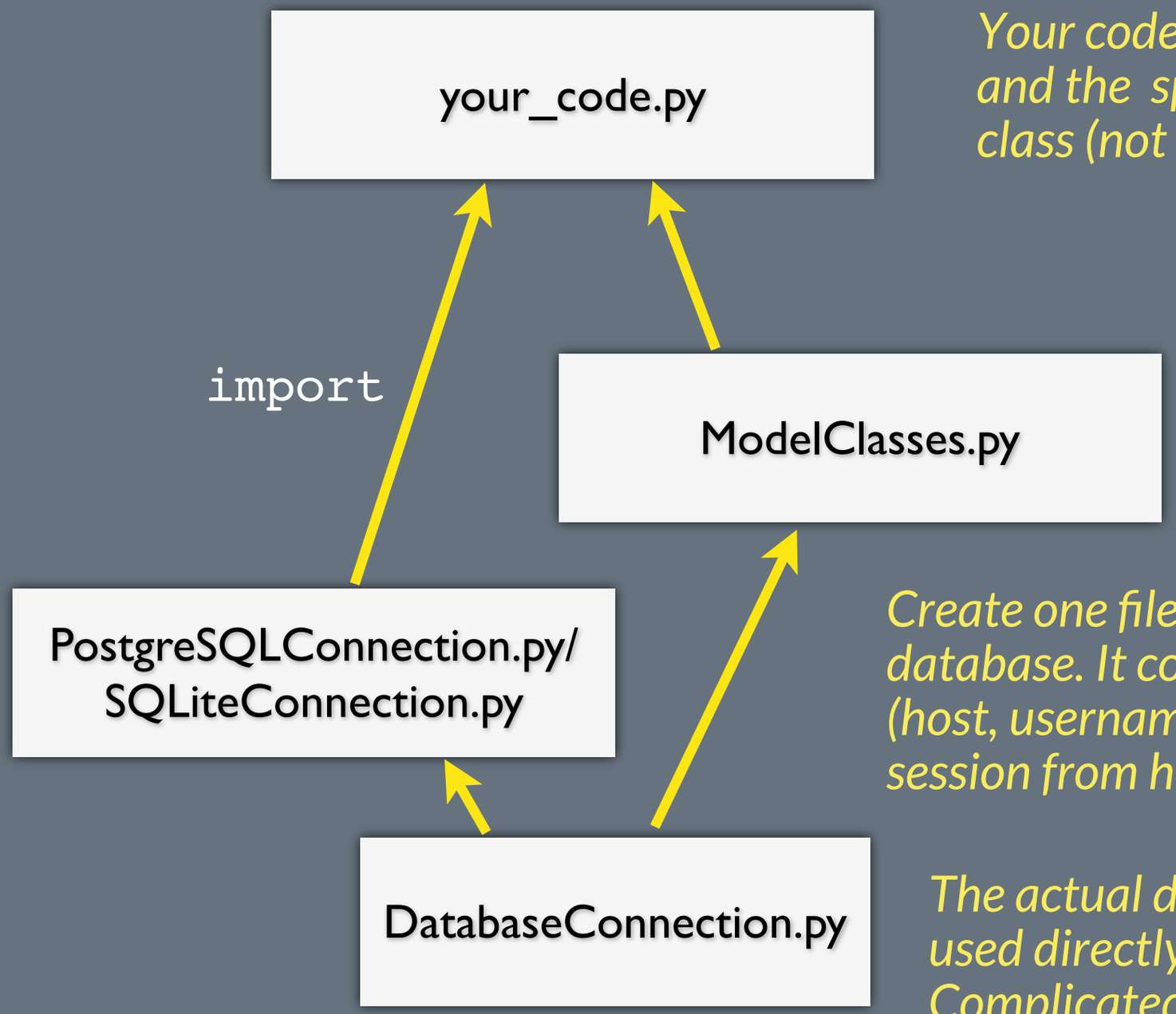
```
student = session.query(Student).first()
supervisor = session.query(Supervisor).first()

student.supervisors.append(supervisor)
session.commit()
```

How would you remove an object from a relation?

Note that the ORM is managing the primary keys, foreign keys, and join table implicitly for you.



Python Files



Your code imports "ModelClasses.py" and the specific database connection class (not "DatabaseConnection.py").

This defines the mapping between the database tables and the custom Python classes.

Create one file for each connection to a database. It contains the connection settings (host, username, password, etc.). Get the session from here.

The actual database connection – not to be used directly. This is a singleton for Complicated Reasons.

And now… code!