Working with Persistent, Related Objects

Object-Relational Mappings (ORMs)

CS61A Spring 2022

MWF 2:00-3:00pm

cs61a.org

vanshaj [at] berkeley [dot] edu

Recall: Data

So far, we've learned how to represent data in our programs as variables:

```
name = "fido"
age = 2
fav_food = "homework"
```

Recall: Objects

We've also learned how to wrap this data in meaningful structures, like classes:

```python
class Animal:
    def __init__(self, name, age, fav_food):
        self.name = name
        self.age = age
        self.fav_food = fav_food
        self.shelter = None # we'll come back to this on the next slide

fido = Animal("fido", 2, "homework")
luna = Animal("luna", 4, "yarn")
```

Recall: Composition

We've even learned how to make objects interact with each other:

```python
class Shelter:
    def __init__(self, name, address, animals):
        self.name = name
        self.address = address
        self.animals = animals
        for animal in animals:
            animal.shelter = self

berkeley = Shelter("Berkeley Shelter", "UC Berkeley", [fido])
```

What are we trying to represent?

Relational data, where each animal is associated with a shelter (and each shelter is associated with some number of animals). We store such data in tables (like you would on paper), and these tables together form a ***database***.

A database is a collection of related data, usually organized in tables, that can be accessed in various ways. The database is generally stored on a computer as a (set of) file(s).

Tables

Animals:

Name	Age	Favorite Food	Shelter
Fido	2	Homework	Berkeley Shelter
Luna	4	Yarn	

A table is a set of data organized into vertical columns (identifiable by name). Each table is usually accompanied by metadata that imposes constraints and relationships on the columns in that table.

Databases

Animals:

Name	Age	Favorite Food	Shelter
Fido	2	Homework	Berkeley Shelter
Luna	4	Yarn	

Shelters:

Name	Address	Animals
Berkeley Shelter	UC Berkeley	[Fido]

Notable Properties

- An `Animal` may or may not be living in a `Shelter`, stored in `animal.shelter`.
- A `Shelter` may have any number of `Animal`s, stored in `shelter.animals`.
- Whenever we change the `shelter` of an `animal`, we need to remove it from the old `shelter.animals` and add it to the new `shelter.animals`.
- Whenever we remove an animal from a `shelter`'s `animals` list, we need to clear the animal's `shelter` attribute.

Broadly: these tables are ***related***, and many actions need to occur simultaneously to keep them in sync.

Many-to-One Relationships

What we've just described is a "many-to-one" relationship: many animals can correspond to one shelter. The converse of this is a "one-to-many" relationship: one shelter can correspond to many animals.

How do we decide which one this is? Well, it's a matter of perspective: Does a shelter have animals? Or do animals each have a shelter?

In our case, it should really be that each animal has a shelter (i.e. lives in a shelter), so we go with many-to-one.

Recall: `Animal`s and `Shelter`s

```python
class Animal:
    def __init__(self, name, age, fav_food):
        ...

class Shelter:
    def __init__(self, name, address, animals):
        ...

fido = Animal("fido", 2, "homework")
luna = Animal("luna", 4, "yarn")
berkeley = Shelter("Berkeley Shelter", "UC Berkeley", [fido])
```

Drawbacks of our Model

How do we maintain a many-to-one relationship? How do we store this across sessions? How do we filter all of our data?

- We have to write our own system for saving this information and restoring it if we want it to persist across sessions.
- We have to write our own syncing system to keep these objects in sync.
- We have to write list comprehensions ourselves if we want to filter/query this data.

Storage

So far, all the programs we've created have saved state on ***memory***, meaning that our data only lasts until we exit the interpreter. Take your Cats high scores, for example -- those don't persist locally, because once you exit the program, your computer gets rid of the information.

To store things long-term, you need to put them in files, such as databases. To share things, you need to put them in some central file that multiple people can access, which is also something that databases allow us to do.

Alternative: ORMs

ORMs (object-relational mappings) wrap databases in a structure that's relevant to the programming language you're working in (for us, that's Python). They let you represent data as related classes, and they handle all the dirty work for you!

In Python, the `sqlalchemy` library provides a set of utilities for working with ORMs.

```
$ pip3 install sqlalchemy
$ python3
>>> import sqlalchemy
>>> sqlalchemy.__version__
'1.4.31'
```

SQLAlchemy: The `Animal` Model

`shelter_system.py`

```python
class Animal(Base):
    __tablename__ = "animals"

    name = Column(String, primary_key=True)
    age = Column(Integer)
    fav_food = Column(String)

    shelter_name = Column(String, ForeignKey("shelters.name"))
    shelter = relationship("Shelter", back_populates="animals")
```

SQLAlchemy: The `Shelter` Model

shelter_system.py

```
class Shelter(Base):
    __tablename__ = "shelters"

    name = Column(String, primary_key=True)
    address = Column(String)


    animals = relationship("Animal", back_populates="shelter")
```

SQLAlchemy Round 1

The following slides will all occur in the same Python interpreter session. This means that, once defined, a variable will continue to exist until we exit the interpreter.

To launch this interpreter with the relevant file loaded in:

```
$ python3 -i shelter_system.py
```

SQLAlchemy 1: Creating Instances

```
>>> fido = Animal(name="fido", age=2, fav_food="homework")
>>> luna = Animal(name="luna", age=4, fav_food="yarn")
>>> fido
Animal(name='fido', age=2, fav_food='homework')
>>> luna
Animal(name='luna', age=4, fav_food='yarn')
>>> fido.shelter
>>> berkeley = Shelter(name="Berkeley Shelter", address="UC Berkeley")
>>> berkeley
Shelter(name='Berkeley Shelter', address='UC Berkeley')
>>> berkeley.animals
[]
```

SQLAlchemy 1: Making our Instances Interact

```
>>> berkeley.animals.append(fido)
>>> berkeley.animals
[Animal(name='fido', age=2, fav_food='homework')]
>>> fido.shelter
Shelter(name='Berkeley Shelter', address='UC Berkeley')
```

```
>>> luna.shelter = berkeley
>>> berkeley.animals
[Animal(name='fido', age=2, fav_food='homework'),
 Animal(name='luna', age=4, fav_food='yarn')]
```

SQLAlchemy 1: Tracking our Objects

```
>>> session.add_all([fido, luna, berkeley])
```

SQLAlchemy 1: Querying our Data

```
>>> session.query(Animal).filter(Animal.shelter == berkeley).all()
[Animal(name='fido', age=2, fav_food='homework'),
 Animal(name='luna', age=4, fav_food='yarn')]
>>> session.query(Animal).filter(Animal.shelter == berkeley,
...                              Animal.age > 3).all()
[Animal(name='luna', age=4, fav_food='yarn')]
>>> fido.shelter = None
>>> session.query(Animal).filter(Animal.shelter == berkeley).all()
[Animal(name='luna', age=4, fav_food='yarn')]
```

SQLAlchemy 1: Saving our Data

```
>>> berkeley
Shelter(name='Berkeley Shelter', address='UC Berkeley')
>>> fido
Animal(name='fido', age=2, fav_food='homework')
>>> luna
Animal(name='luna', age=4, fav_food='yarn')
>>> session.commit()
```

SQLAlchemy Round 2

The following slides will all occur in the same Python interpreter session. This means that, once defined, a variable will continue to exist until we exit the interpreter.

To launch this interpreter with the relevant file loaded in:

```
$ python3 -i shelter_system.py
```

SQLAlchemy 2: Loading our Data

```
>>> berkeley = session.query(Shelter).one()
>>> berkeley
Shelter(name='Berkeley Shelter', address='UC Berkeley')
>>> berkeley.animals
[Animal(name='luna', age=4, fav_food='yarn')]
>>> luna = berkeley.animals[0]
>>> luna
Animal(name='luna', age=4, fav_food='yarn')
>>> fido = session.query(Animal).filter(Animal.shelter == None).one()
>>> fido
Animal(name='fido', age=2, fav_food='homework')
```

SQLAlchemy 2: Modifying and Saving our Data

```
>>> fido.shelter = berkeley
>>> session.dirty
IdentitySet([
    Animal(name='fido', age=2, fav_food='homework'),
    Shelter(name='Berkeley Shelter', address='UC Berkeley')
])
>>> session.commit()
```

SQLAlchemy Round 3

The following slides will all occur in the same Python interpreter session. This means that, once defined, a variable will continue to exist until we exit the interpreter.

To launch this interpreter with the relevant file loaded in:

```
$ python3 -i shelter_system.py
```

SQLAlchemy 3: Loading our Data

```
>>> session.query(Shelter).one().animals
[Animal(name='fido', age=2, fav_food='homework'),
 Animal(name='luna', age=4, fav_food='yarn')]
```

All of our data persists! And we didn't have to write a syncing mechanism, *or* a save/load system, *or* a filtering system. SQLAlchemy takes care of all of this for us, which saves us a ton of time when our models get more complicated, connected, involved, and numerous.

Examples of Real-World Use: OH Queue

```python
class User(db.Model, UserMixin):
    __tablename__ = "user"
    id = db.Column(db.Integer, primary_key=True)
    created = db.Column(db.DateTime, default=db.func.now())
    email = db.Column(db.String(255), nullable=False, index=True)
    name = db.Column(db.String(255), nullable=False)
    is_staff = db.Column(db.Boolean, default=False)

    course = db.Column(db.String(255), nullable=False, index=True)

    call_url = db.Column(db.String(255))
    doc_url = db.Column(db.String(255))
```

Examples of Real-World Use: OH Queue

```
class Ticket(db.Model):
    __tablename__ = "ticket"
    id = db.Column(db.Integer, primary_key=True)
    created = db.Column(db.DateTime, default=db.func.now(), index=True)
    updated = db.Column(db.DateTime, onupdate=db.func.now())
    status = db.Column(EnumType(TicketStatus), index=True)

    sort_key = db.Column(db.DateTime, default=db.func.now(), index=True)

    user_id = db.Column(db.ForeignKey("user.id"), index=True)
    helper_id = db.Column(db.ForeignKey("user.id"), index=True)
    assign_id = db.Column(db.ForeignKey("assignment.id"), index=True)
    ...
```

Examples of Real-World Use: OH Queue

```
class Ticket(db.Model):
    ...
    location_id = db.Column(db.ForeignKey("location.id"), index=True)
    question = db.Column(db.String(255))
    description = db.Column(db.Text)

    user = db.relationship(User, foreign_keys=[user_id])
    helper = db.relationship(User, foreign_keys=[helper_id])
    assignment = db.relationship(Assignment, foreign_keys=[assign_id])
    location = db.relationship(Location, foreign_keys=[location_id])

    course = db.Column(db.String(255))
```

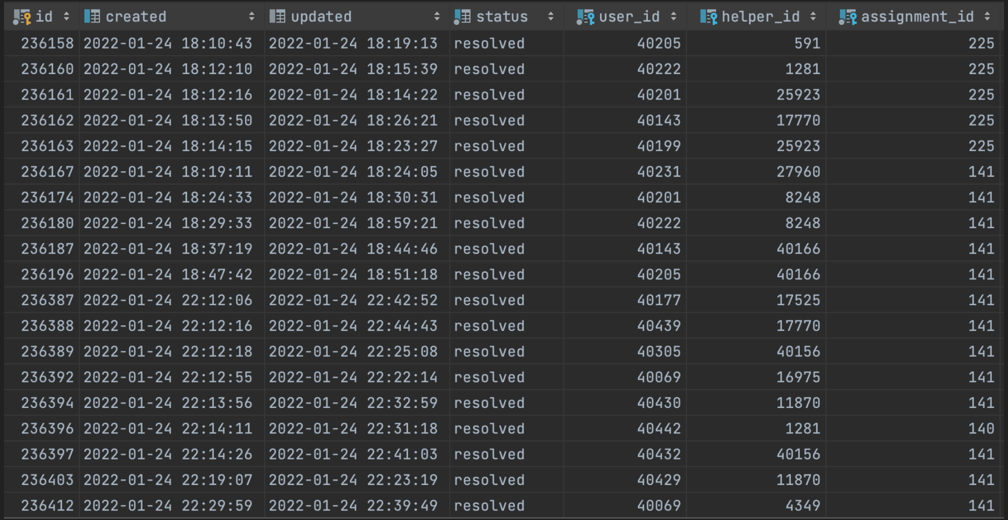
Examples of Real-World Use: OH Queue

```python
class TicketEvent(db.Model):
    __tablename__ = "ticket_event"
    id = db.Column(db.Integer, primary_key=True)
    time = db.Column(db.DateTime, default=db.func.now())
    event_type = db.Column(EnumType(TicketEventType), nullable=False)
    ticket_id = db.Column(db.ForeignKey("ticket.id"), nullable=False)
    user_id = db.Column(db.ForeignKey("user.id"), nullable=False)

    course = db.Column(db.String(255), nullable=False, index=True)

    ticket = db.relationship(Ticket)
    user = db.relationship(User)
```

Examples of Real-World Use: OH Queue

Example of some data you might get out of the OH Queue:

```
>>> session.query(Ticket).filter(Ticket.course == "cs61b")
```



id	created	updated	status	user_id	helper_id	assignment_id
236158	2022-01-24 18:10:43	2022-01-24 18:19:13	resolved	40205	591	225
236160	2022-01-24 18:12:10	2022-01-24 18:15:39	resolved	40222	1281	225
236161	2022-01-24 18:12:16	2022-01-24 18:14:22	resolved	40201	25923	225
236162	2022-01-24 18:13:50	2022-01-24 18:26:21	resolved	40143	17770	225
236163	2022-01-24 18:14:15	2022-01-24 18:23:27	resolved	40199	25923	225
236167	2022-01-24 18:19:11	2022-01-24 18:24:05	resolved	40231	27960	141
236174	2022-01-24 18:24:33	2022-01-24 18:30:31	resolved	40201	8248	141
236180	2022-01-24 18:29:33	2022-01-24 18:59:21	resolved	40222	8248	141
236187	2022-01-24 18:37:19	2022-01-24 18:44:46	resolved	40143	40166	141
236196	2022-01-24 18:47:42	2022-01-24 18:51:18	resolved	40205	40166	141
236387	2022-01-24 22:12:06	2022-01-24 22:42:52	resolved	40177	17525	141
236388	2022-01-24 22:12:16	2022-01-24 22:44:43	resolved	40439	17770	141
236389	2022-01-24 22:12:18	2022-01-24 22:25:08	resolved	40305	40156	141
236392	2022-01-24 22:12:55	2022-01-24 22:22:14	resolved	40069	16975	141
236394	2022-01-24 22:13:56	2022-01-24 22:32:59	resolved	40430	11870	141
236396	2022-01-24 22:14:11	2022-01-24 22:31:18	resolved	40442	1281	140
236397	2022-01-24 22:14:26	2022-01-24 22:41:03	resolved	40432	40156	141
236403	2022-01-24 22:19:07	2022-01-24 22:23:19	resolved	40429	11870	141
236412	2022-01-24 22:29:59	2022-01-24 22:39:49	resolved	40069	4349	141

Examples of Real-World Use: Sections Tool

```python
class Section(db.Model):
    id: int = db.Column(db.Integer, primary_key=True)
    course: str = db.Column(db.String(255), index=True)
    description: str = db.Column(db.String(255))
    capacity: int = db.Column(db.Integer)
    can_self_enroll: bool = db.Column(db.Boolean)
    enrollment_code: str = db.Column(db.String(255), nullable=True)
    staff_id: int = db.Column(db.Integer, db.ForeignKey("user.id"))
    staff: "User" = db.relationship(
        "User",
        backref=db.backref("sections_taught", lazy="joined"),
        foreign_keys=[staff_id],
    )
```

Examples of Real-World Use: Sections Tool

```python
class Slot(db.Model):
    id: int = db.Column(db.Integer, primary_key=True)
    course: str = db.Column(db.String(255), index=True)
    name: str = db.Column(db.String(255))
    start_time: int = db.Column(db.Integer)
    end_time: int = db.Column(db.Integer)
    location: str = db.Column(db.String(255), nullable=False)
    call_link: str = db.Column(db.String(255), nullable=True)
    section_id: int = db.Column(db.Integer, db.ForeignKey("section.id"))
    section: Section = db.relationship(
        "Section", backref=db.backref("slots")
    )
    sessions: List["Session"]
```

Examples of Real-World Use: Sections Tool

```python
class Session(db.Model):
    id: int = db.Column(db.Integer, primary_key=True)
    course: str = db.Column(db.String(255), index=True)
    start_time: int = db.Column(db.Integer)
    slot_id: int = db.Column(db.Integer, db.ForeignKey("slot.id"))
    attendances: List["Attendance"]

class AttendanceStatus(Enum):
    present = 1
    excused = 2
    absent = 3
```

Examples of Real-World Use: Sections Tool

```python
class Attendance(db.Model):
    id: int = db.Column(db.Integer, primary_key=True)
    course: str = db.Column(db.String(255), index=True)
    status: AttendanceStatus = db.Column(db.Enum(AttendanceStatus))
    session_id: int = db.Column(db.Integer, db.ForeignKey("session.id"))
    session: Session = db.relationship(
        "Session", backref=db.backref("attendances", lazy="joined"),
        innerjoin=True,
    )
    student_id: int = db.Column(db.Integer, db.ForeignKey("user.id"))
    student: "User" = db.relationship(
        "User", backref=db.backref("attendances"), innerjoin=True
    )
```

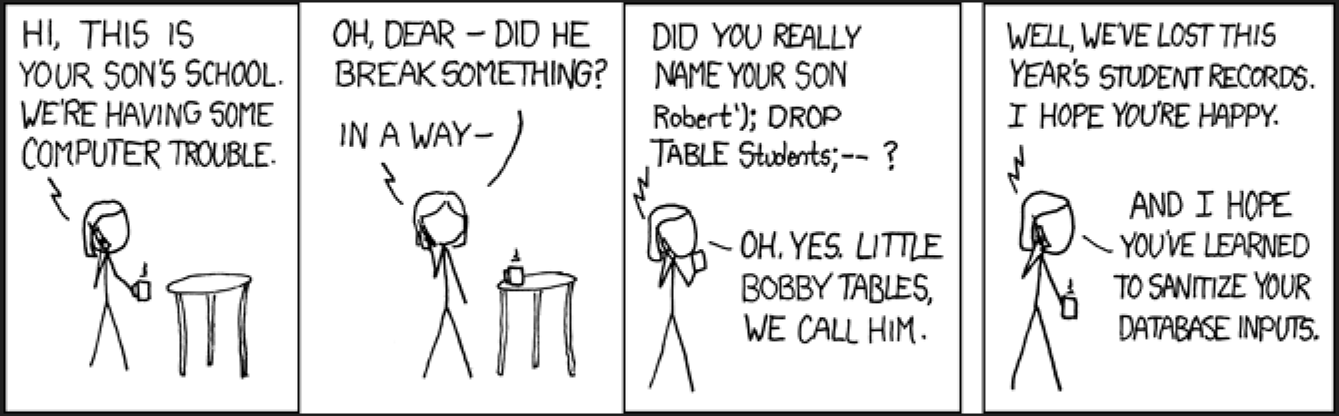
(Appendix) Another Possible Solution: SQL

SQL (Structured Query Language) is a declarative language (more on these after Spring Break) that is designed to help manage large amounts of (often related) data. We used to teach this in 61A after Scheme, but in a way that was pretty detached from the rest of the course.

Biggest drawback: learning a new language with wildly different syntax.

```
SELECT * FROM animals WHERE age > 3 AND shelter = "Berkeley Shelter"
    AND fav_food = "yarn";
SELECT * FROM shelters WHERE name = "Berkeley Shelter"
    AND address = "UC Berkeley";
```

(Appendix) Problems with SQL: Security


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

The exploit above is known as SQL injection, and you can learn more about it by taking CS 161 or CS 169A.

(Appendix) Problems with SQL: Unwieldy

```
select student.section_id as section_id,
       student.email      as student_email,
       student.name       as student_name,
       slot.name          as section_type,
       slot.location      as section_location,
       staff.email        as staff_email,
       staff.name         as staff_name
from user as student, slot, user as staff, section
where student.course = "cs61a"
  and student.is_staff = 0
  and student.section_id = slot.section_id
  and student.section_id = section.id
  and section.staff_id = staff.id
```

(Appendix) SQLAlchemy: Setting Up

To start using SQLAlchemy, we need to install and import it, then set up a `Base` class that our models will inherit from. You don't really need to understand this code, beyond its purpose (hooking into a local database at `shelters.db`).

`shelter_system.py`

```python
from sqlalchemy import create_engine
from sqlalchemy import Column, String, Integer, ForeignKey
from sqlalchemy.orm import declarative_base, sessionmaker, relationship

engine = create_engine("sqlite:///shelters.db", echo=False)
Base = declarative_base()
```

(Appendix) SQLAlchemy: Declaring the Models

Once we've created all of the subclasses of `Base` that we wanted, it's time to declare them in our database and create a `session` that we can use to read from/write to our database.

`shelter_system.py`

```python
Base.metadata.create_all(engine)

Session = sessionmaker(bind=engine)
session = Session()
```

Submit anonymous feedback at imvs.me/t/anon

Thanks for stopping by :)

vanshaj [at] berkeley [dot] edu