Python Django

Peter Williams
Matthew Fulmer

What is Django?

- Object-relational mapper
  - Data models are defined entirely in Python
- Supports OO idioms
  - Inheritance (interface + subclassing)
- Database support
  - Oracle, MySQL, PostgreSQL, SQLite
- Powerful admin features
  - Provides a great GUI for examining the database, making changes, etc.
  - More of a novelty for programmers who know what they're doing/have background with databases
- Django also has a lot of useful features for rapid website development, among other things, but we don't really care about them for this course

Models + Fields

- Model
  - Python class representing a single database table
  - Must be a subclass of the built-in django.db.models.Model
  - Can define model methods (row-level functionality)
  - Can be auto-generated from an existing, legacy database
- Fields
  - Attributes of Model classes
  - Represent columns in the data table
  - There are many built-in field types, or you can write your own (more on this soon)
  - They all correspond to a data table type (INTEGER, VARCHAR, etc)
  - Plenty of options to set
    - max_length, primary key, configuration for null or empty values, etc

Models + Fields (continued)

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()
```

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id"),
    "choice" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
COMMIT;
```

Models + Fields (continued)

```
from django.contrib.localflavor.us.models import USStateField

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()
    address = models.CharField(max_length=100)
    city = models.CharField(max_length=50)
    state = USStateField() # Yes, this is America-centric...

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if datetime.date(1945, 8, 1) <= self.birth_date <= datetime.date(1964, 12, 31):
            return "Baby boomer"
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        return "Post-boomer"

    def is_midwestern(self):
        "Returns True if this person is from the Midwest."
        return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')
```

Custom Fields

- Comparable to PostgreSQL's custom types
- Model class doesn't store Fields for its attributes (it just stores regular Python objects)
- Field class stored in the Model's Meta class, *used for converting between the Python object and the database value*
- Methods must be written to:
  - Convert the database value to a Python object
  - Convert a Python object to a format suitable for a parameter in a database query
  - Convert a query value to a database value
  - Define how to handle database lookups (exact, contains, >, <, startswith, etc)

Many-to-Many

- Simpler than Hibernate
  - Only need to define the relationship (create a ManyToManyField) in one of the 2 models involved
  - Don't have to set it on both sides
- However, each side has slightly different syntax
  - Ex: Pizza has a ManyToManyField for Toppings
  - p1.toppings.all() vs. t1.pizza_set.all()

Many-to-Many (with intermediate Model)

- What if you want to associate data with the relationship between 2 models?

```
class Person(models.Model):
    name = models.CharField(max_length=128)

    def __unicode__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __unicode__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

- Assignment and .add() do not work
  - Need to create an instance of the intermediate model

QuerySets

- Allows complex query construction using filter(), exclude(), get(), group_by, having, aggregation etc.

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

- Use F() expressions to reference the current value of a model field within a query

```
Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

- QuerySets are lazy!
  - DB is only hit once (when?)

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.now())
>>> q = q.exclude(body_text__icontains="food")
>>> print q
```

- Python array-slicing is the equivalent of SQL LIMIT

Database Interaction

- Saving - Model.save()
  - Checks to see if object has a primary key set
  - If so, calls a SELECT to see if it's in the table
  - If so, performs UPDATE, if not, or if primary key was not set, performs INSERT
- Deletion - Model.delete()
  - Does not remove the Python objects (just calls DELETE on row in the db)
  - Can also use on QuerySets
  - Emulates ON DELETE CASCADE behavior by default

Transactions

- By default runs an open transaction that commits automatically and immediately when save(), delete(), etc are called
  - No implicit ROLLBACK
- Can also set to commit on success to require manual commit and abort conditions

Performance: DB Access Optimization

- Don't forget the obvious
  - Use indexes where needed
  - Use field types appropriately and efficiently
- Understand QuerySets
  - QuerySets cache the results, so use iterators instead of list()
    - List() loads every object into memory at once, iterators load the objects one-at-a-time
- Understand how the cache works
- You can use raw SQL if you need to

Questions?