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)

```python
from django.db import models

class Mail(models.Model):
    subject = models.CharField(max_length=200)
    text = models.TextField()
    sent_date = models.DateField(auto_now_add=True)

    def __str__(self):
        return self.subject

# Create your models here
```

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)

```python
class CustomField(models.Field):

    def __init__(self, *args, **kwargs):
        self.lookup_type = kwargs.pop('lookup_type', 'exact')
        super(CustomField, self).__init__(*args, **kwargs)

    def get_prep_lookup(self, value):
        if self.lookup_type == 'exact':
            return self.to_python(value)
        elif self.lookup_type == 'contains':
            return self.to_python(value)

    def to_python(self, value):
        # Convert the Python object to the database value
        return value

# Example usage

class MyModel(models.Model):
    custom_field = CustomField(max_length=100, lookup_type='contains')

    def __str__(self):
        return self.custom_field
```

```python
# Example usage

my_model = MyModel(custom_field='example')
my_model.save()
```
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?
- 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.
  ```python
  QuerySet().filter(some_field__isnull=False)
  ```
- Use F() expressions to reference the current value of a model field within a query
  ```python
  QuerySet().filter(objects__some_field=F('other_model__field'))
  ```
- QuerySets are lazy!
  - DB is only hit once (when?)
  ```python
  q = query_set.queryset
  ```
- 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

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?