Exercise 1. From Django model to database.

Ex 1a. Discussion forum.

We'll build a simple discussion forum for users to write comments on various topics. Each comment belongs to a single topic, and is authored by one user.

Create a project called **`discuss`**, and an app called **`forum`**. Configure it with postgres, using a fresh database name that you have not used before. Use "create database discuss' on psql, and use "discuss" as the name of the databse. Edit **`models.py`** and create the following models. There is no primary key here.

User
    name: varchar (30 chars) # possible to have name clashes.

Topic
    title: varchars (200 chars) blank=False, null=False
    created_by: User (foreign key to User)
    created_on: datetime (see auto_now_add on DateTimeField

Comment:
    topic: Topic foreignkey
    created_by: User foreignkey field
    subject: varchar(200 chars) # must not be empty
    message: varchar(1000 chars)

Ex 1b. Build the database and test drive it.

1. Add app forum to settings.py
2. `python3 manage.py makemigrations`
3. `python3 manage.py migrate`
4. psql connect to database. Using "\d topic", see what fields have been added to the topic table. What is their definition? Similarly for the other tables (Answer this for yourself)
5. Add `'django_extensions'` to settings.py:INSTALLED_APPS.
6. `python3 manage.py shell_plus`

7. Create two user instances ("santa", "banta"), two topics ("politics", "eggs") , and two comments per post on the shell prompt. You can assign them to any user, but make sure every user has said something.
8. Retrieve topics that have the title "politics". There should be one member in this QuerySet.
   Here's a useful cheatsheet for all commands.
   https://github.com/chrisdl/Django-QuerySet-Cheatsheet
   The Django documentation is excellent. See:
   https://docs.djangoproject.com/en/3.0/topics/db/queries/
9. Retrieve all comments under topic "eggs"
10. Retrieve all comments on all topics made by users that have the names "nta" in them. This should retrieve all comments.

Ex 1d. Write automatic tests with the above queries.

Edit tests.py to do the above checks in an automatically testable form. Then run `python manage.py test`. See https://docs.djangoproject.com/en/3.0/topics/testing/overview/ for how to write a test.

1. test_topic_search: filter for all topics with the title politics. Use the various forms of python's unittest.assert… methods to assert that the search did indeed retrieve the topics you entered.
2. test_comments_per_topic: filter for all comments that belong to topic 'eggs' and assert that they are as expected
3. test_comments_by_selected_users: All comments made by users with the string 'nta' in their usernames, and ensure that the count is as expected.

Each testcase must have a def setup(self) method (see link above) where you create all the model instances that you need while testing. Django wipes the db clean before the test case starts, for repeatability

Ex 1c. Adding an admin interface.

In admin.py simply add the following lines, one per model

```
from django.contrib import admin
admin.site.register(Topic)
admin.site.register(Comment)
admin.site.register(User)
```

This automatically generates a default admin UI for each model. See localhost:/8000/admin. You should already see the data you entered there.

Submit this project as your_roll_number-ex1-feb11.zip

Exercise 2. From database to model: working backwards from a legacy database

In this exercise we are going to take an existing ("legacy") database and see what it takes to make it work with Django. We will work with a subset of the university database outlined in Prof. Sudarshan's book.

It turns out that Django has a few constraints.

1. Every model must have a primary key. This is what makes it object-oriented. By default, Django creates an "id" primary key column for each model.
2. **Django does not support composite keys** (many columns together to form a primary key). The university schema from Prof. Sudarshan's book has many tables with composite keys, such as section, classroom, and some that do not have any primary keys at all. We'll be using a subset of the schema as shown below (The labels on the arrows are the names used on both sides of the foreign relationships).

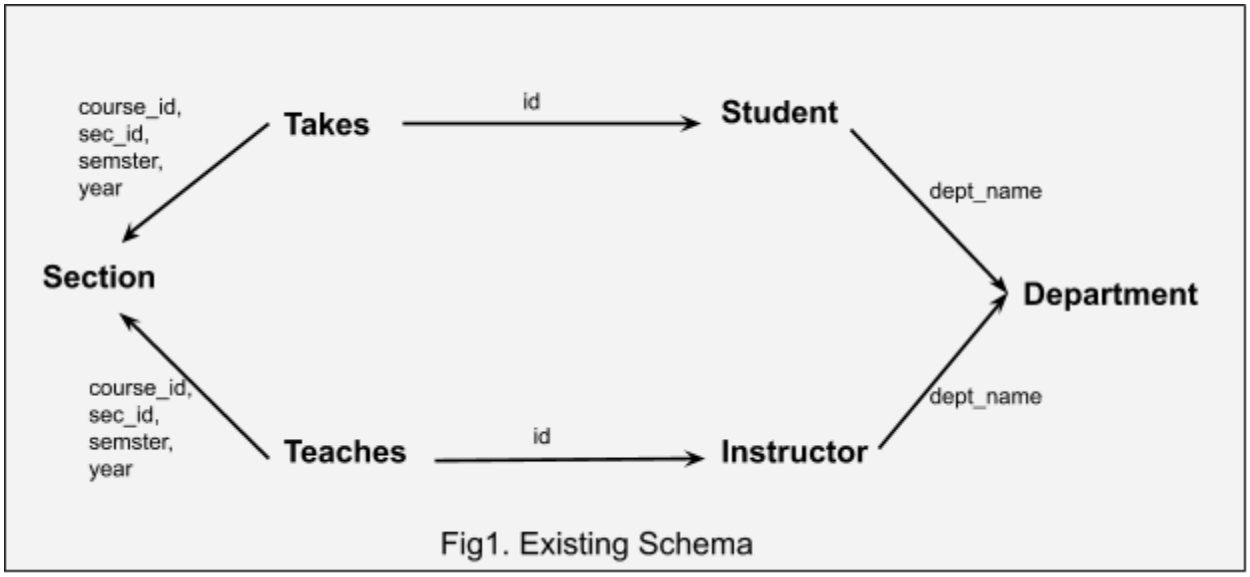

Fig1. Existing Schema

3. Third, consider the takes and teaches relations. Takes does not have a primary key, since it is a many-many relationship. However it needs one to fit in the django world. Further, it has a field already called "id", which only confuses Django. So we'll rename the columns takes.id to takes.student_id and teaches.id to teaches.instructor_id

respectively. Figure 2 shows the desired changes.

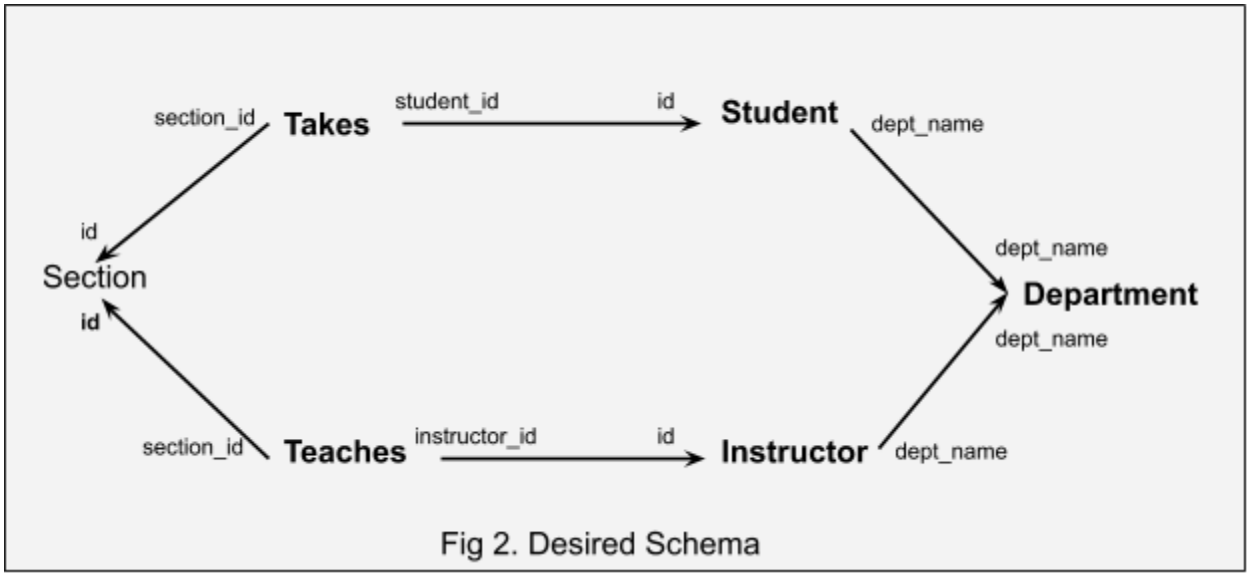

Fig 2. Desired Schema

Exercise 2a. Django'ize the database

1. Download smallRelationsInsertFile.sql from this assignment on moodle and load the data using psql. Load the data in a database called "uni"
2. The following is an example of what it takes to change one of the relationships, between takes and section. We construct the new arrow using new columns, drop the old arrow, then formalize the new arrow as a foreign key reference. **All the schema changes must be done using "alter table".** If in doubt about the syntax, use "\h alter table" at the psql prompt. Collect the following steps in a .sql file (see submission guideline below).

    a. Add columns takes:section_id (type integer) and section:id (type serial).
    b. Fill takes.section_id up with the corresponding section.id, by matching the course,sec,semester year, to preserve the relationship between the tuples in the two tables.
    c. In *takes*, drop foreign key constraint referencing section. Do this with the *teaches* relation as well (otherwise we can't do the next step)
    d. Drop *section*'s primary key constraint
    e. Make id the primary key constraint for section
    f. Add foreign key constraint on takes that refers takes.section_id to section.id
    g. We may as well drop the course,sec,semester, year columns on takes, since we will not be using it.
    h. Put a unique constraint on section.(course_id,sec_id,year, semester). Earlier it was unique combination by dint of being a primary key

3. Repeat this exercise with all the other solid arrows. There is nothing to be done with the dotted arrows yet. Remember to rename takes.id to takes.student_id first (likewise with teaches). Only then can you add an id column on teakes and teaches.

Put all these steps in your_rollnumber-ex2-feb11.sql file. At the end you will combine it with the exercise 2's zip file.

Ex 2b. Data → Django Model

We have the data in the database, and now we are automatically going to generate python model code from the db.

1. Set up a new django project uni, with an app called acad. Configure db and installed_apps settings. Don't run makemigrations yet.

2. `python3 manage.py inspectdb > acad/models.py`. This automatically generates the models. The model and database schemas are quite close, but we need to tweak it a little bit for a slightly better interaction.
3. Remove the managed=False line for each mode. This tells Django that it is the manager of the data, not a passive user.
4. For all relationship (arrow) fields, add a "related_name" parameter. See fig 3 below. The name of the field represents the model's end of the arrow. The "related_name" names the other end of the arrow. For example, student refers to department as follows

   ```
   dept = models.ForeignKey(Department,
                   models.DO_NOTHING,
                   db_column='dept_name', blank=True, null=True,
                   related_name="students")
   ```

   This way, student.dept refers to the department, and department.objects.students_at refer to all the students in that department. The related_name represents the arrow head, and can be used as a virtual field to traverse that relationship in reverse (e.g. physicsdept.students)

5. Django has nice support for Many to Many relationships, such as teaches and takes and advisor. Instructor and Section are connected via Teaches, which we can formalize in Django. See the dotted lines in fig 3. For example, in class Instructor, add

   ```
   section = models.ManyToManyField(Section,
   ```

```
through='Teaches',
through_fields=('instructor','section'),
related_name="taught_by")
```

Note the through and through_fields and refer to Fig 2 A many-many field needs to be mentioned in only one of the classes in the relationship. Now we can retrieve objects from either side of this dotted relationship, without having to explicitly join in the intermediate table. Django does that under the covers.

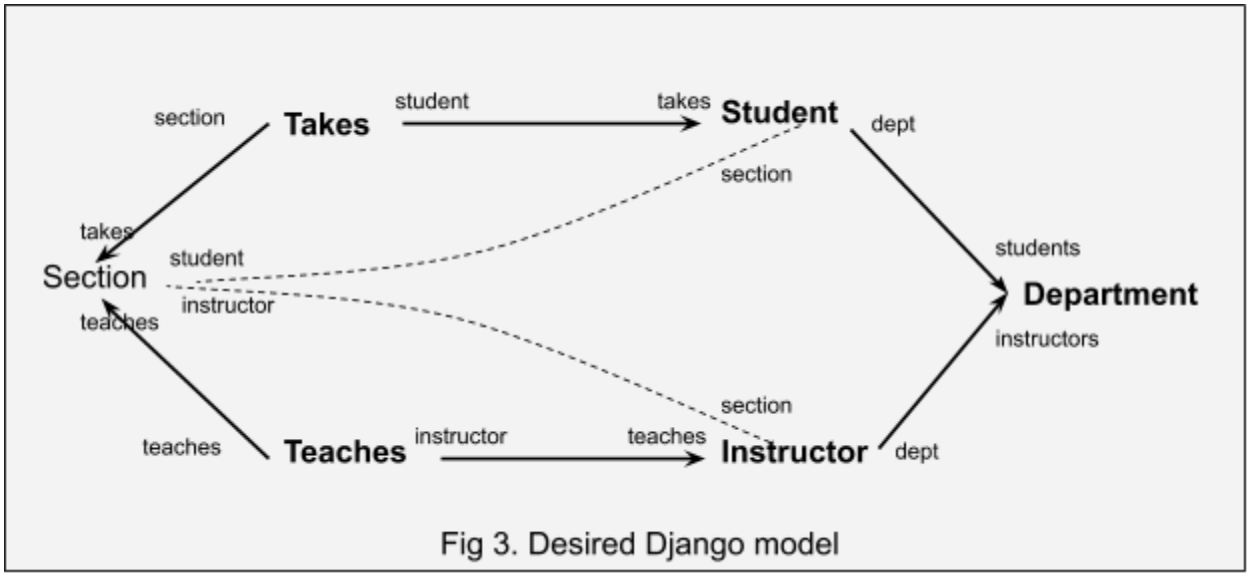

Fig 3. Desired Django model

4. Now don't do makemigrations. Just call 'python3 manage.py migrate'. If you have done makemigrations, it is fine. Search for files under the migration folder, which is the set of sql that django plans to run on the database. It is fine to delete those files. The problem is that if you don't,, Django will attempt to create the tables, but those tables already exist.

Ex 2c. Test the model.

Test your model in tests.py . We already have loads of test data to work with. Write a single test case with multiple and load the smallRelationsinsertFile as part of your test setup.
In each test case, we are going to compare the data we have received with the data from a raw sql query, and assert that the data is the same. You can get access to the database connection object used by django.

For example, a test case could be like this:

```
from django.db import connection
def test_student_count(self):
       (header, rows) = db_exec(connection, "select count(*) from student")
       sql_student_count = rows[0][0]
       model_student_count = Student.object.count()
```

```
        self.assertEquals(sql_student_count, model_student_count)
```

Note that queries don't *have* to be one-liners, but it is always better to push the computation to the database as far as you can.

Recollect that Django creates a test database every single time. So in your tests.py, your **setUp**() function should load the data from the main database to the test_database. Note that you can't load the smallRelationsinsertFile.sql now because the db schema has changed. Hint: Look at pg_dump to dump the data from the main database, and replay the file in your test db.

Write a series of test cases to help learn about querying Django models, including selection, aggregation, traversal of related objects.

The queries are:

1. The number of students in the "Physics" department.
2. The number of students in the "Physics" and "Comp. Sci." department. Concept: *Q expression* syntax in filter.
3. The list of student names who are in the departments located in the 'taylor' building. Do an ignore-case match. Concept: *iexact*, *values_list*. You can specify only the attributes you want in value_list
4. The total money each department spends on instructor salaries, as a list ordered by department name. Concept: *annotate* each department with the aggregate sum .
5. The list of course names taken by a student named 'Tanaka'. This should be a flat list.
6. A list of all the names of instructors that a student name 'Brown' has taken courses with. This involves traversing student, section, and instructor via their intermediate m-n relationships. Make sure an instructor is mentioned only once. *hint: Use the __in filter*

Move the your_rollnumber-ex2-feb11.sql file from earlier into the uni project directory (at the same level as manage.py)
Upload this project as your_rollnumber-ex2-feb11.zip to moodle