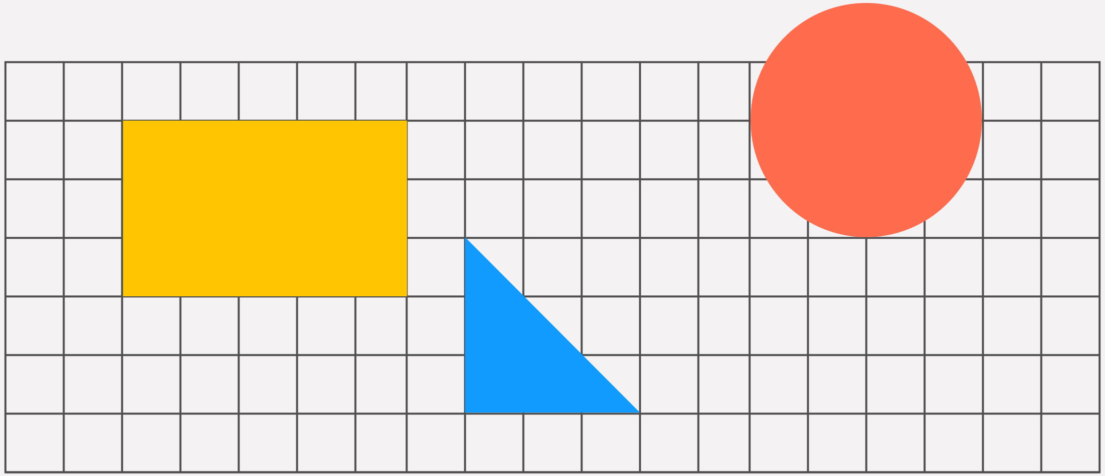
Django Models

Ali Abrishami



In this Lecture...

a

Define and Structure Database Tables

b

Learn Object-Relational Mapper

c

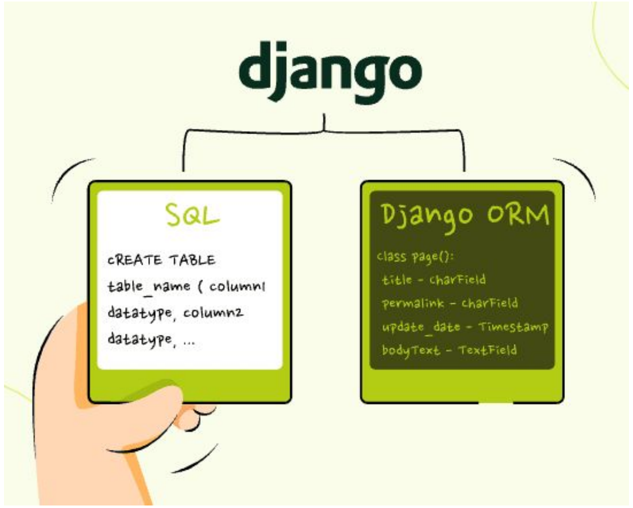
Explore Django's Admin Panel

What is a Model?

- A model represents the data structure of your application. It's the "shape" of the **information** you store.
- It defines your fields and relationships, describing how pieces of data connect to each other.
- It acts as Django's **abstraction layer** over the database, letting you think in **objects** instead of SQL tables.
- It becomes the single source of truth for data: used to generate tables, enforce rules, and manage persistence.

Django Model

- Each model maps to a single database table and defines:
  - **Fields** (database columns)
  - **Behaviors** (methods)
  - **Relationships** (between tables)
  - **Metadata** (table options)

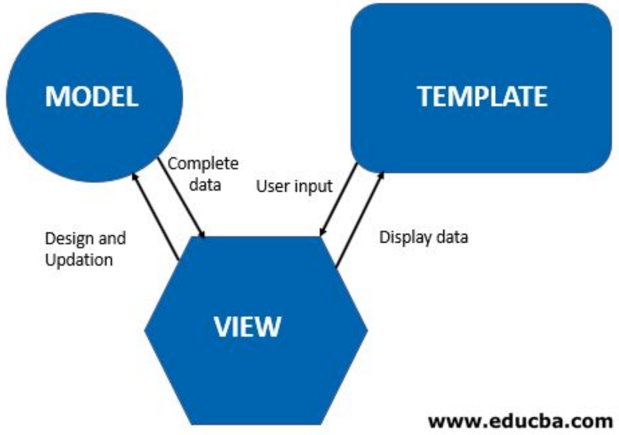


```
cREATE TABLE
table_name ( column1
datatype, column2
datatype, ...
```

```
class page():
 title - charField
 permalink - charField
 update_date - Timestamp
 bodyText - TextField
```

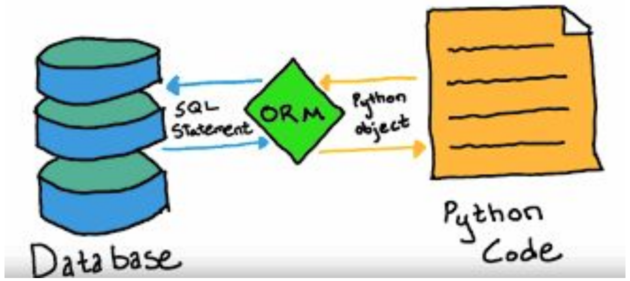
The M in MVT

- The **Model** is Django's layer for handling data: defining **structure**, **constraints**, and **relationships**.
- It encapsulates the business logic related to data.
- It provides a database-agnostic interface.
  - letting Django talk to the DB through ORM.
- It acts as the **foundation** other layers rely on:
  - Views retrieve and manipulate it.
  - Templates display it.



ORM!

- An ORM is a bridge between code and database, letting you work with **objects** instead of raw SQL.
- It translates Python operations (create, query, update, delete) into **database commands** behind the scenes. And It enforces type-safety and structure, based on your model definitions.
- It reduces boilerplate and mistakes by giving you a consistent, database-independent API.



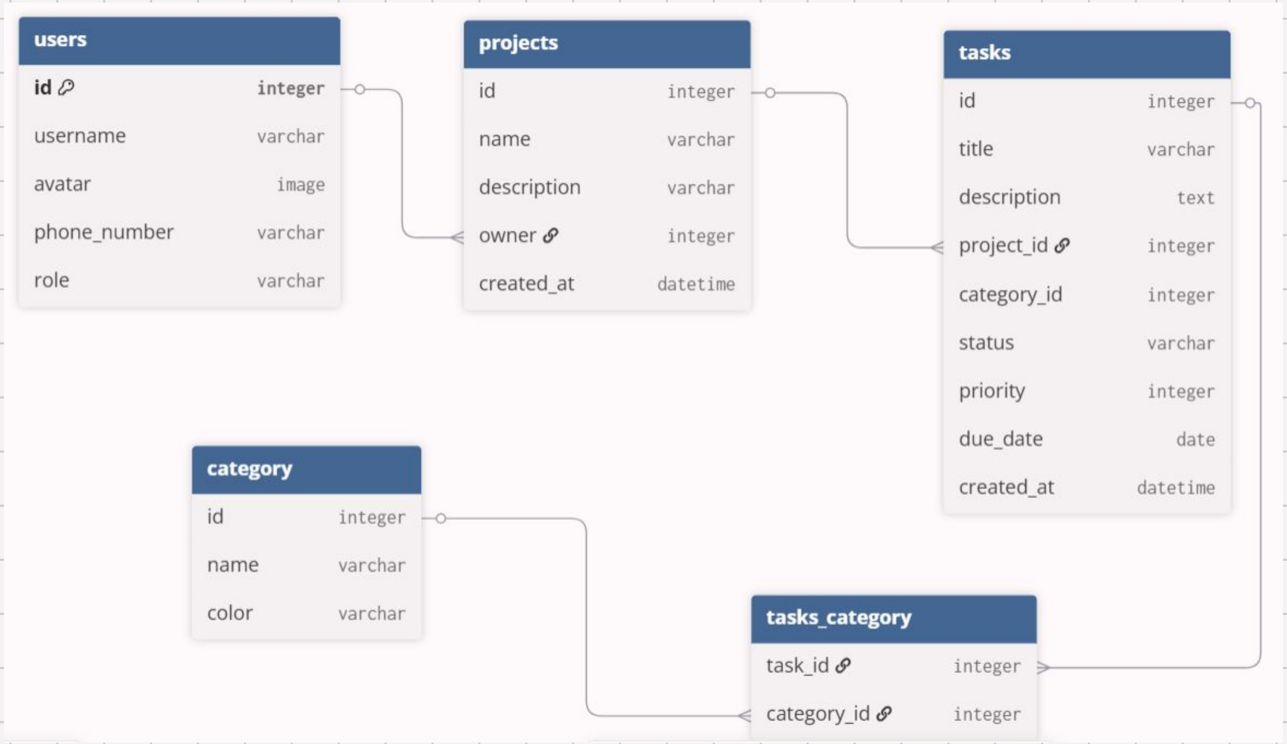
DB Configuration

- Django defines its database settings in `settings.py` under the `DATABASES` dictionary, where you specify engine, name, host, port, and credentials.
- These settings control which database backend Django uses and how it connects to it.
- Django reads this config at startup and routes all ORM operations through the defined backend.
- Changing the database only requires updating this configuration, your model code stays the same.

SQLite!

- SQLite is a lightweight, file-based database included by default with Python and Django.
- It stores all data in a single `.sqlite3` file, requiring no separate server or setup.
- Perfect for development, small projects, and testing, but limited for heavy concurrency and large-scale apps.
- Django uses SQLite as its default database engine to allow instant project bootstrapping with zero configuration.

Step 1: Design Your Models



users	
id	integer
username	varchar
avatar	image
phone_number	varchar
role	varchar

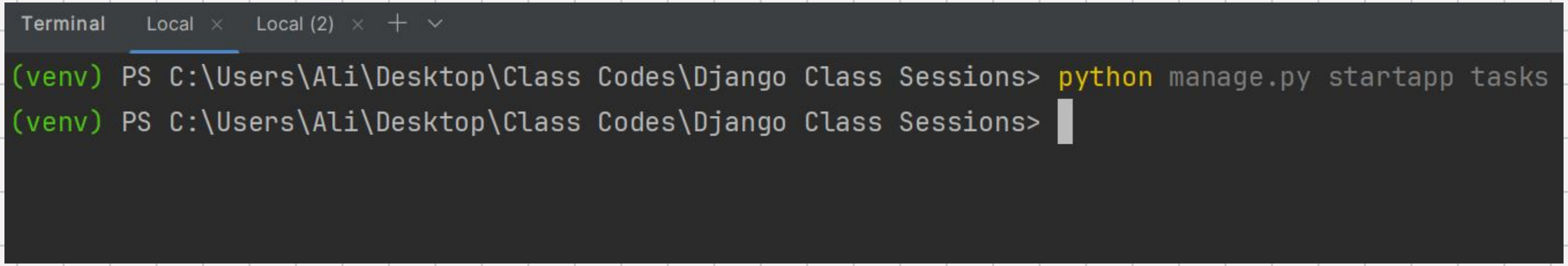
projects	
id	integer
name	varchar
description	varchar
owner	integer
created_at	datetime

tasks	
id	integer
title	varchar
description	text
project_id	integer
category_id	integer
status	varchar
priority	integer
due_date	date
created_at	datetime

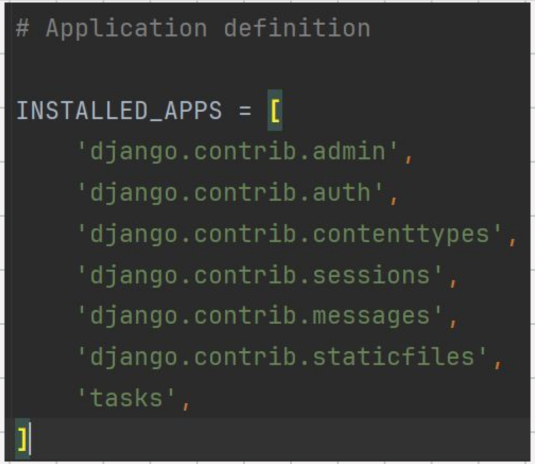
category	
id	integer
name	varchar
color	varchar

tasks_category	
task_id	integer
category_id	integer

Step 2: Create Your Apps



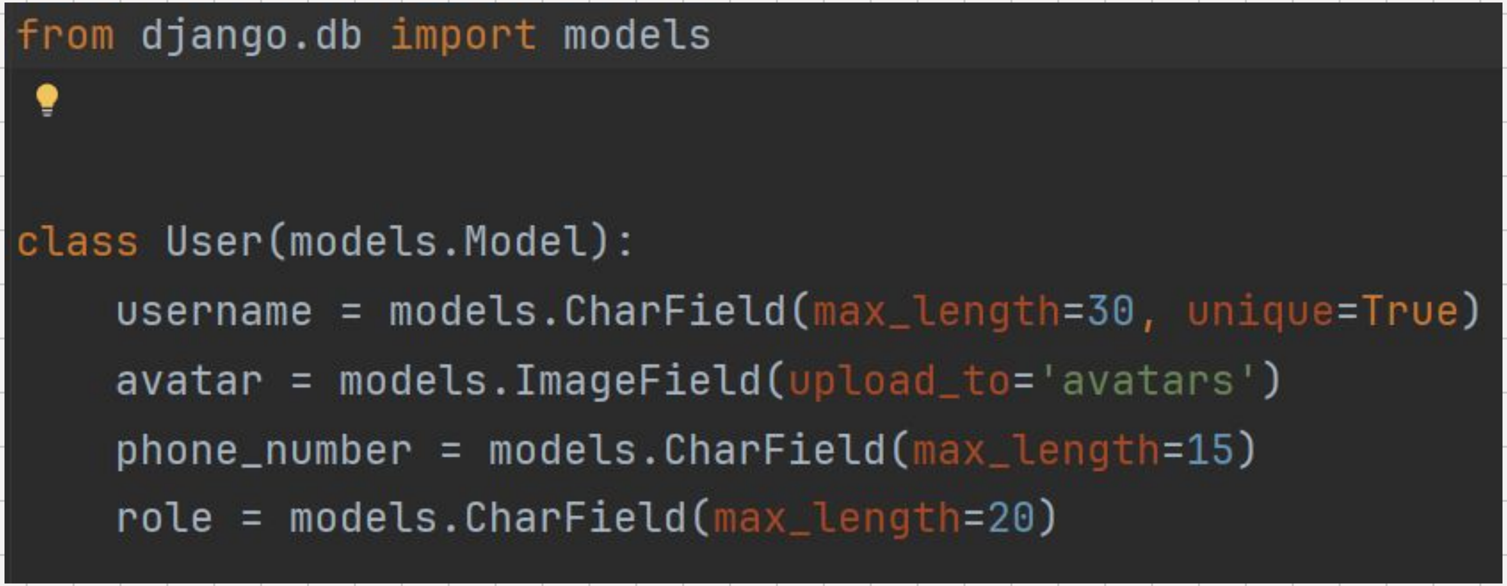
```
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions> python manage.py startapp tasks
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions>
```



```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'tasks',
]
```

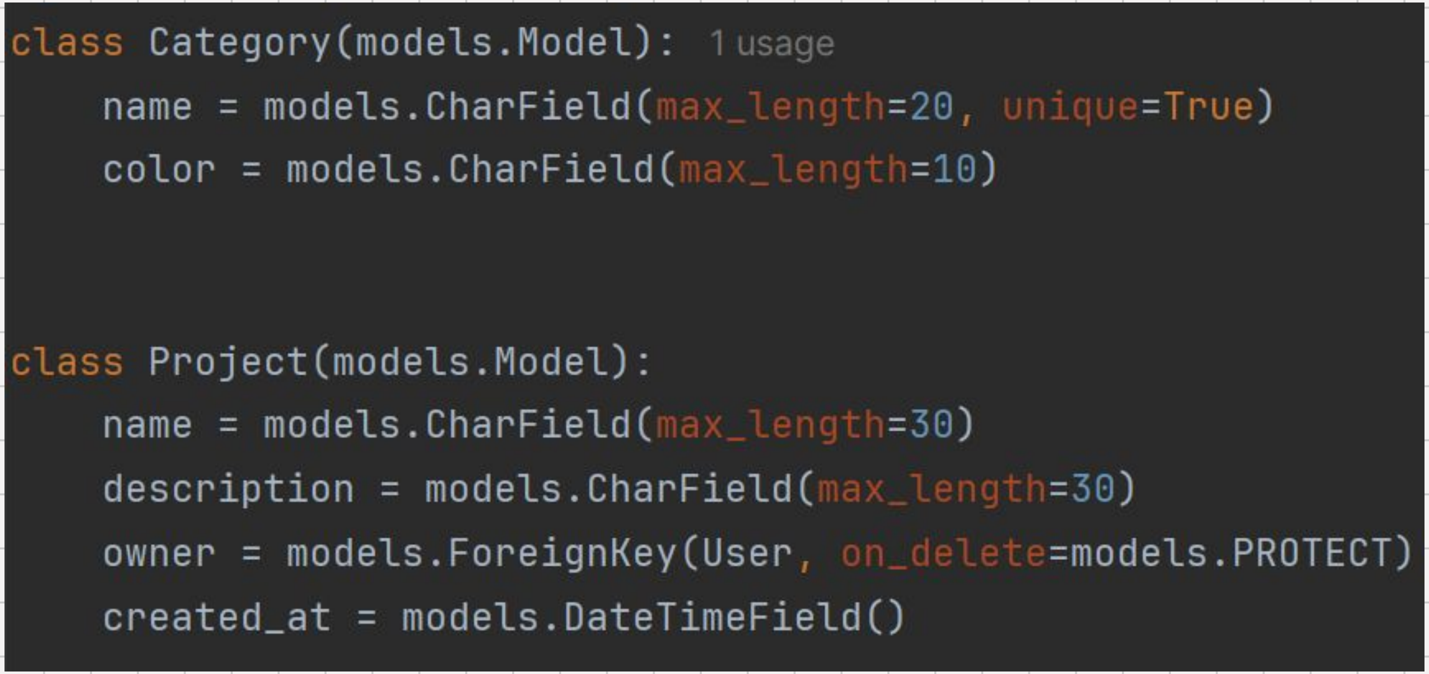
Create tasks Models



```
from django.db import models


class User(models.Model):
    username = models.CharField(max_length=30, unique=True)
    avatar = models.ImageField(upload_to='avatars')
    phone_number = models.CharField(max_length=15)
    role = models.CharField(max_length=20)
```

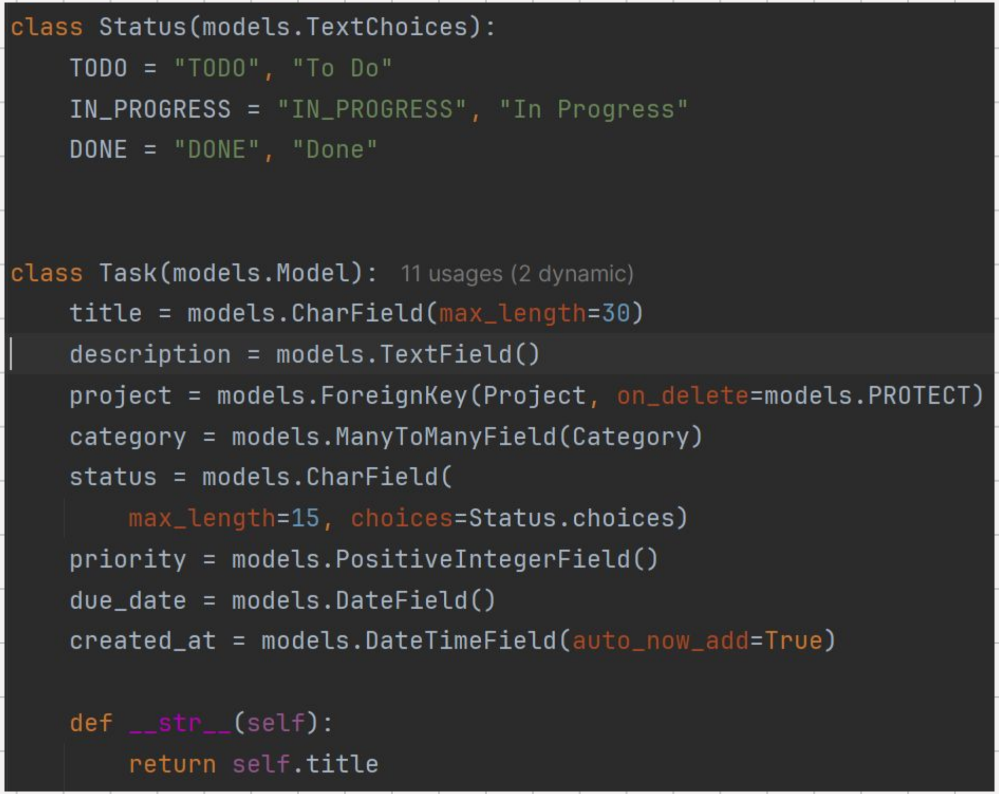
Create tasks Models (Cont.)



```
class Category(models.Model):  1 usage
    name = models.CharField(max_length=20, unique=True)
    color = models.CharField(max_length=10)


class Project(models.Model):
    name = models.CharField(max_length=30)
    description = models.CharField(max_length=30)
    owner = models.ForeignKey(User, on_delete=models.PROTECT)
    created_at = models.DateTimeField()
```

Create tasks Models (Cont.)



```
class Status(models.TextChoices):
    TODO = "TODO", "To Do"
    IN_PROGRESS = "IN_PROGRESS", "In Progress"
    DONE = "DONE", "Done"


class Task(models.Model):  11 usages (2 dynamic)
    title = models.CharField(max_length=30)
    description = models.TextField()
    project = models.ForeignKey(Project, on_delete=models.PROTECT)
    category = models.ManyToManyField(Category)
    status = models.CharField(
        max_length=15, choices=Status.choices)
    priority = models.PositiveIntegerField()
    due_date = models.DateField()
    created_at = models.DateTimeField(auto_now_add=True)

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

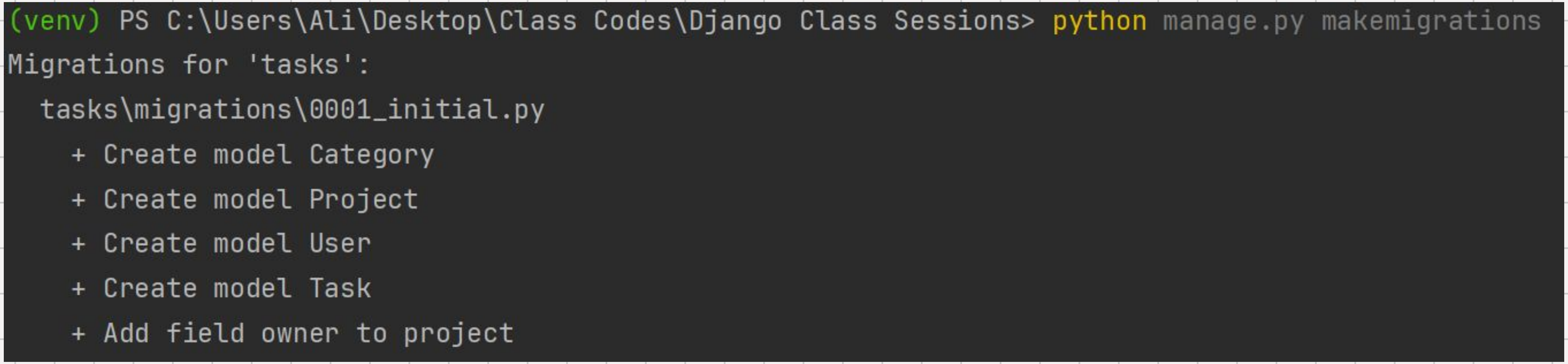
What is Migration?

Django migration is a **version control system** for your database schema. It's how Django:

1. **Tracks** changes to your models (adding/removing fields, tables)
2. **Applies** those changes to your database without writing SQL manually.
3. **Rolls back** changes if needed

Create tasks Models (Cont.)

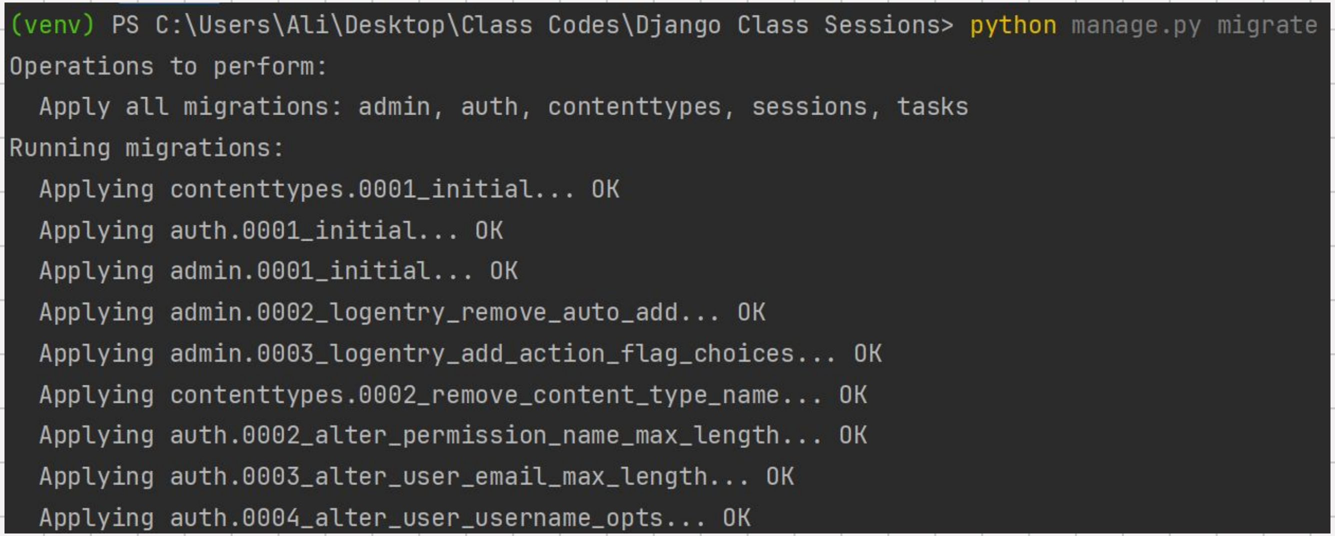
In the end, you have to run `makemigrations`, and then `migrate`



```
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions> python manage.py makemigrations
Migrations for 'tasks':
  tasks\migrations\0001_initial.py
    + Create model Category
    + Create model Project
    + Create model User
    + Create model Task
    + Add field owner to project
```

Create tasks Models (Cont.)

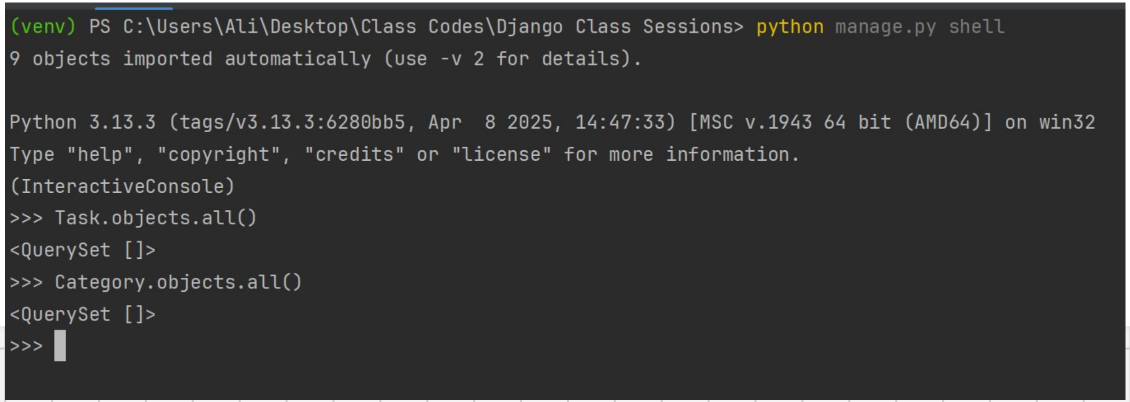
In the end, you have to run `makemigrations`, and then `migrate`



```
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, tasks
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
```

Explore Django Shell

- Using `python manage.py shell`, it opens an interactive environment with full project context for experimenting safely.
- It lets you create, query, filter, update, and delete model instances directly.
- It helps you inspect generated SQL, isolate issues, and confirm ORM behavior without touching the app.



```
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions> python manage.py shell
9 objects imported automatically (use -v 2 for details).

Python 3.13.3 (tags/v3.13.3:6280bb5, Apr  8 2025, 14:47:33) [MSC v.1943 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> Task.objects.all()
<QuerySet []>
>>> Category.objects.all()
<QuerySet []>
>>> 
```

Common model fields

- **Text-based fields** for storing names, titles, descriptions, and long-form content.
- **Numeric fields** for integers, decimals, counters, and IDs.
- **Boolean fields** for simple true/false data.
- **Date and time fields** for timestamps, scheduling, and auto-tracking creation/updates.
- **Relational fields** for linking models together through one-to-one, many-to-one, or many-to-many relationships.

Django Model Fields

Field	Usage
`CharField`	**Short** text (e.g. name, text, title)
`TextField`	**Long** text (e.g. content, description, biography)
`IntegerField`	**Whole** numbers
`FloatField`	Decimal numbers with **less precision**
`DecimalField`	Decimal numbers with **more precision**
`BooleanField`	**True** or **false** values

Django Model Fields (cont.)

Field	Usage
`DateField`	**Date-only** field
`DateTimeField`	**Date** and **time**
`TimeField`	**Time-only** field
`EmailField`	**Email** address with **validation**
`UrlField`	**URL** string with **validation**
`SlugField`	**URL**-friendly **short label**

Django Model Fields (cont.)

Field	Usage
`UUIDField`	**Unique identifier** (UUID)
`AutoField`	**Auto-incrementing primary key**
`JSONField`	Stores **structured JSON** data
`ImageField`	For **image** uploads
`FileField`	For **file** uploads
`BinaryField`	Stores **raw binary** data

Field Options

- Django model fields also come with options.
- These options are specified using arguments passed to the field.
- Field options help limit user input or set default values.
  - You can allow a field to be null by setting the appropriate option.
  - You can also create database indexes for faster lookups.
- Each time you change a field option, you need to create a new migration (more on that later).

Field Options (cont.)

Option	Usage
`null=True`	Allow database **NULL** (for **non-required fields**)
`blank=True`	Allow **empty input** in forms
`default=...`	Set a **default** value
`choices=...`	**Limit** input to **specific options** (Best used with `Choices`)
`unique=True`	Ensure all values for this field are **unique**
`primary_key=True`	Specify as **primary key**

Field Options (cont.)

Option	Usage
`auto_now`	Set **current time** on **every save**
`auto_now_add`	Set **current time** once when **created**
`max_length`	**Limit** string **length** (`CharField`, `TextField`, etc.)
`upload_to`	Define **path** for uploaded **files or images**
`unique_for_date`	Value is **unique** within a **date**
`editable=False`	**Exclude** from **admin panel** and **forms**

Important Options to know

- **On delete:** Defines what happens to related objects when the referenced object is **deleted** (Example: `on_delete=models.CASCADE` deletes dependent rows automatically).

- **Upload to:** Specifies the subdirectory inside `MEDIA_ROOT` where uploaded files/images are stored (Example: `upload_to='profile_pics/'` saves uploads to `/media/profile_pics/`).

- **Null vs. Blank:**
  - `null=True`: Database can store `NULL` (used for non-string fields).
  - `blank=True`: Field is allowed to be empty in forms (used for validation).

- **Note:** For string-based fields, prefer `blank=True, null=False`.

CRUD Operations

- **Create**: generating new model instances and saving them to the database through the ORM.
  - `Product.objects.create(...)`
  - `Product.objects.bulk_create(...)`
  - `Product.objects.get_or_create(...)`

- **Read**: retrieving data using filters, lookups, and `QuerySet`s to fetch exactly what you need.
  - `Product.objects.all()`
  - `Product.objects.get(id=1)`
  - `Product.objects.first()`
  - `Product.objects.filter(...)`

CRUD Operations (cont.)

- **Update**: modifying existing records and saving changes back to the database.
  - `product.save()`
  - `Product.objects.filter(...).update(...)`
  - `Product.objects.update_or_create(...)`

- **Delete**: removing records cleanly and safely using the model's delete mechanisms.
  - `product.delete()`
  - `Product.objects.filter(...).delete()`

Queryset fundamentals

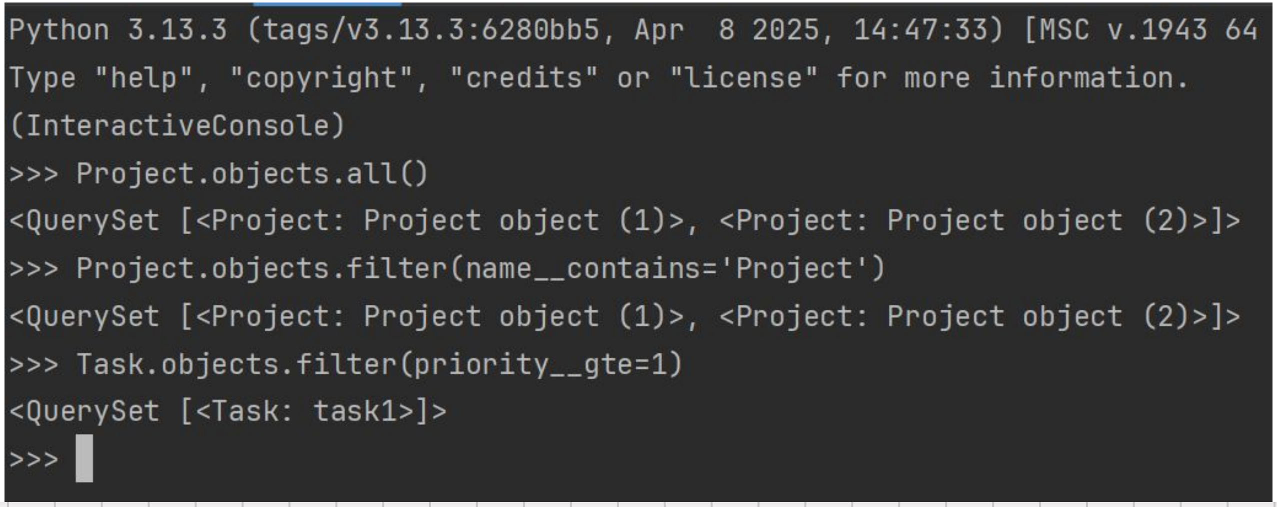
- A QuerySet is a collection of database queries that retrieve objects from your database.
- It represents a **lazy** database lookup—queries aren't executed until **needed**.
- **Key Characteristics:**
  - **Lazy evaluation**: Query executes only when needed (e.g., iteration, slicing, printing).
  - **Chained operations**: Methods like .filter() return new QuerySets and can be chained.
  - **Immutable**: Each QuerySet call creates a new QuerySet; existing ones aren't modified.
  - **Optimized/Efficient**: Django minimizes database hits (e.g., via lazy loading, caching results after first evaluation).

Filtering & Querying

- `filter()`, `get()`, and `exclude()` shape which records you retrieve.
- Filters can be chained to progressively narrow down results.
- Field lookups (`contains`, `exact`, `gte`, `lte`, `startswith`, etc.) give precision when querying.
- `get()` returns exactly one object, while `filter()` returns a `QuerySet` even when empty.

Field lookups

- Field lookups are how you specify the meat of an SQL WHERE clause.
- They're specified as keyword arguments to the QuerySet methods filter(), exclude() and get().



```
Python 3.13.3 (tags/v3.13.3:6280bb5, Apr  8 2025, 14:47:33) [MSC v.1943 64
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> Project.objects.all()
<QuerySet [<Project: Project object (1)>, <Project: Project object (2)>]>
>>> Project.objects.filter(name__contains='Project')
<QuerySet [<Project: Project object (1)>, <Project: Project object (2)>]>
>>> Task.objects.filter(priority__gte=1)
<QuerySet [<Task: task1>]>
>>>
```

Ordering & Slicing

- QuerySets support ordering results by one or more fields in **ascending** or **descending** order.
- Slicing behaves like Python list slicing but triggers a LIMIT/OFFSET query under the hood.
- Ordering and slicing are both lazily combined until evaluation.
- Complex ordered queries remain database-level operations, not in-memory sorting.
    - `Product.objects.filter(...).order_by(‘-date’)[:5]`

Aggregations

- Aggregation functions (`Count`, `Avg`, `Sum`, `Max`, `Min`) compute values across QuerySets.
- They run single SQL aggregation queries, not Python loops.
- Useful for analytics, summaries, or performance-critical calculations.
- Aggregations collapse results into dictionaries or values instead of returning QuerySets.
  - `Book.objects.aggregate(Avg("price", default=0))`
  - `Book.objects.all().aggregate(Avg("price"))`
- Read more at https://docs.djangoproject.com/en/5.2/topics/db/aggregation/

Annotating QuerySets

- Annotations attach calculated fields to each row in the QuerySet.
- Useful for per-object computed metrics like totals, differences, or conditional counts.
- These calculations happen at the database level, not in Python.
- Allows mixing raw values and annotated expressions in filters, ordering, and templates.

```
author_summary = Author.objects.annotate(
      total_books=Count('book'),
      average_price=Avg('book__price')
)
```

Quick Review on Relations

- **One-to-One:** Each record in one table is linked to exactly one record in another table; this type of relationship is used when each side of the relationship should hold unique data (Example: A user has one profile, and each profile belongs to one user).

- **Many-to-One:** Many records from one table can be associated with one record in another table; this is a common relationship in relational databases (Example: Many students belong to one university, and each university can have many students).

- **Many-to-Many:** Records from one table can relate to multiple records in another table, and vice versa; this usually requires a junction table to manage associations (Example: Students enroll in many courses, and each course can have many students).

Model Relationships

- Three types:
  - `ForeignKey` (1-many)
  - `ManyToManyField`
  - `OneToOneField`
- `related_name` and reverse lookups control how related objects are accessed from both sides.
- Many-to-many relationships rely on an intermediate table, auto-managed or custom.

on_delete Option

- `on_delete` determines how child objects react when a parent is removed.
    - `CASCADE`: Delete this object when the referenced object is deleted (default)
    - `PROTECT`: Prevent deletion of referenced object if related objects exist
    - `RESTRICT`: Similar to PROTECT, but with different database-level constraints
    - `SET_NULL`: Set this field to NULL when referenced object is deleted
    - `SET_DEFAULT`: Set this field to its default value when referenced object is deleted
    - `SET(...)`: Set this field to specific value or callable when referenced object is deleted
    - `DO_NOTHING`: Take no action (database-level constraints may cause errors)

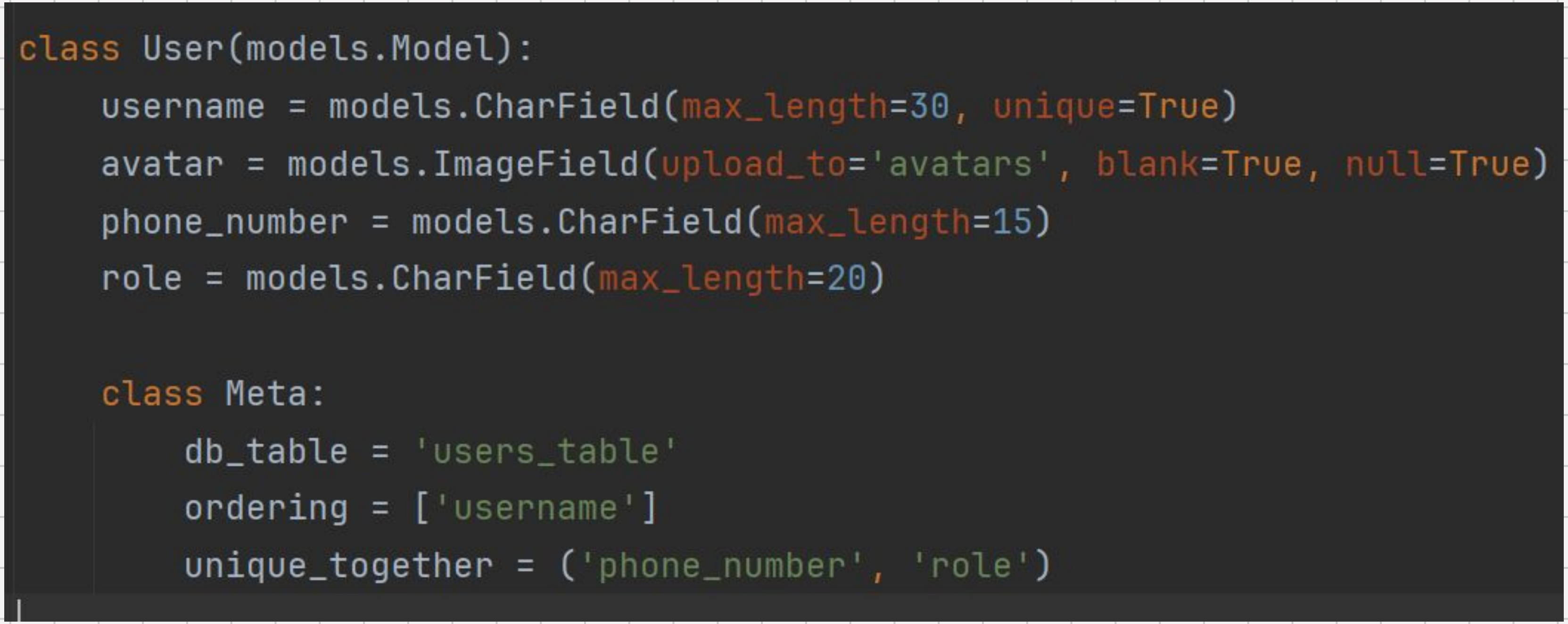
Django Models Tips

- **Use `__str__` wisely**: Return a meaningful string to represent each object in the admin and shell.
- **Set `related_name`**: Customize reverse relationships to avoid confusing defaults.
- **Use `choices` for enums**: This keeps your data consistent and readable.
- **Add indexes for search fields**: Use `db_index=True` on fields you filter or sort frequently.
- **Use Meta options**: Set `ordering`, `verbose_name`, and `unique_together` to fine-tune behavior.
- **Don't forget migrations**: Run `makemigrations` and `migrate` after every model change to apply it.

Meta!

- The `Meta` class lets you define **model-level configuration** separate from fields.
- Controls behavior such as **ordering**, database **table name**, and **permissions**.
- Enables options like making a model **abstract** or setting **unique constraints**.
- Provides fine-grained tuning of how Django interacts with the model at the ORM and DB level.

Sample Meta Class



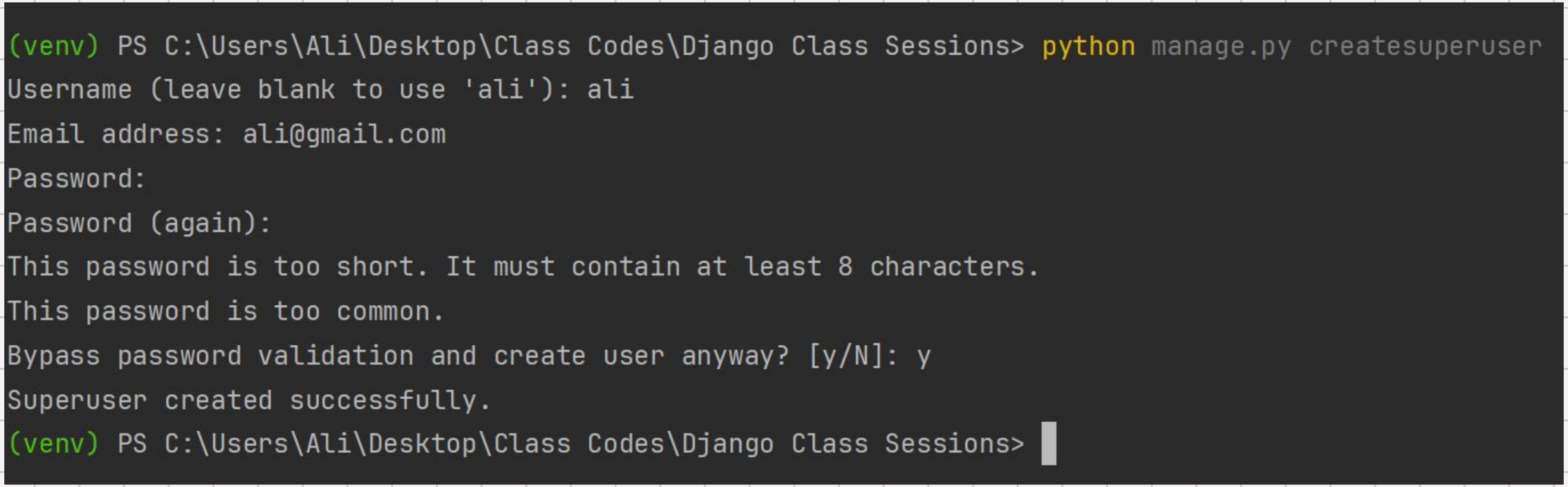
```python
class User(models.Model):
    username = models.CharField(max_length=30, unique=True)
    avatar = models.ImageField(upload_to='avatars', blank=True, null=True)
    phone_number = models.CharField(max_length=15)
    role = models.CharField(max_length=20)

    class Meta:
        db_table = 'users_table'
        ordering = ['username']
        unique_together = ('phone_number', 'role')
```

Introducing Django Admin

- Django Admin gives you a **web interface** to **manage** your models with almost no setup.
- By default, it's at `/admin/` of your application.
- To register your models, use `admin.site.register(Model)` this makes models appear in the admin panel.
- Use `list_display`, `search_fields`, and `list_filter` to control how data shows.
- To start, you need to create an admin user. To do this, you should use `createsuperuser` command with `manage.py`.

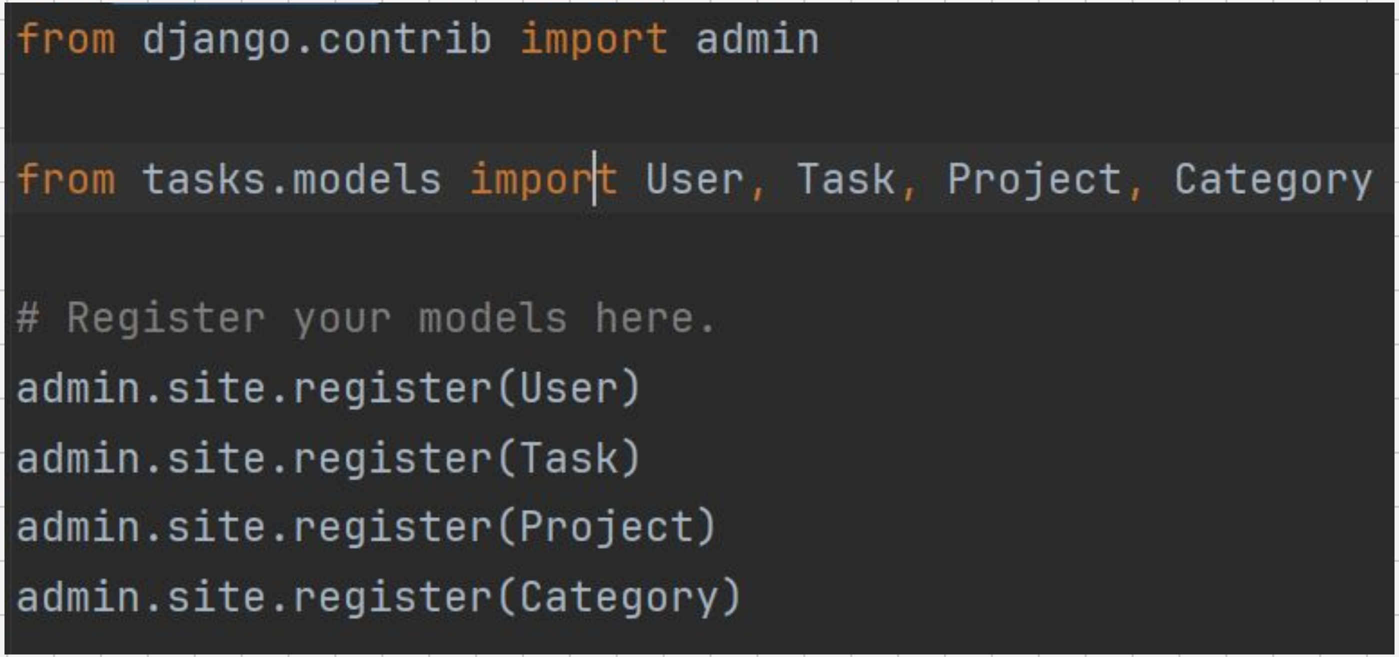
Creating The Admin User



```
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions> python manage.py createsuperuser
Username (leave blank to use 'ali'): ali
Email address: ali@gmail.com
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions>
```

`createsuperuser` asks you some questions, then creates a new admin/superuser based on the information you provide.

Register The Models

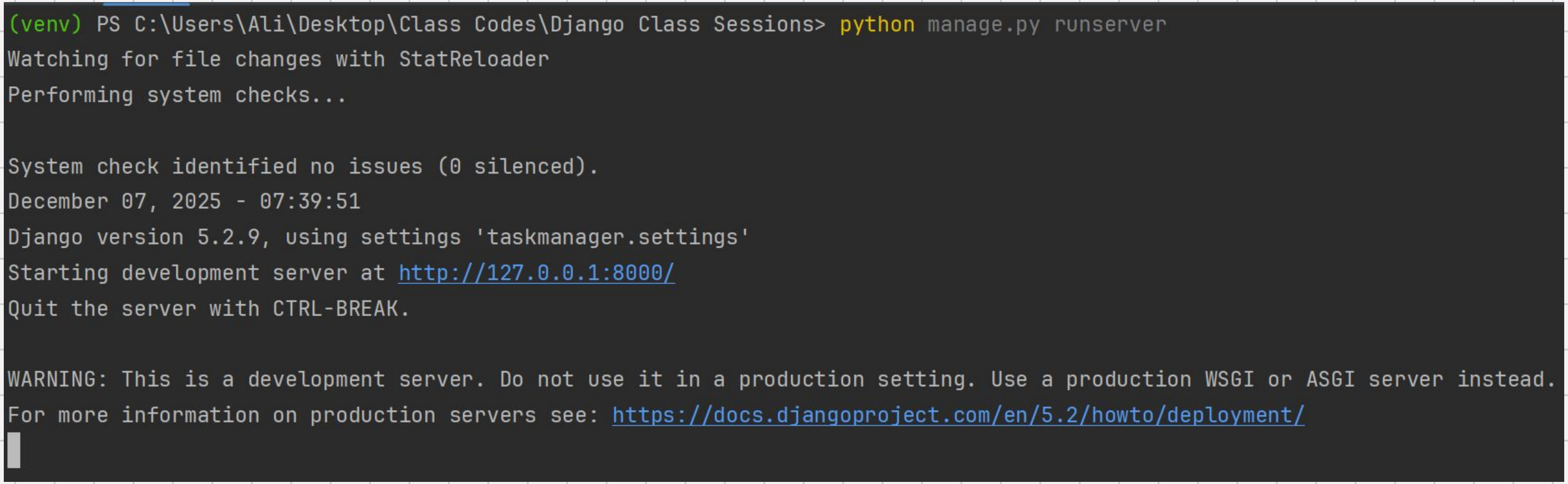


```python
from django.contrib import admin

from tasks.models import User, Task, Project, Category

# Register your models here.
admin.site.register(User)
admin.site.register(Task)
admin.site.register(Project)
admin.site.register(Category)
```

Run Your Django App

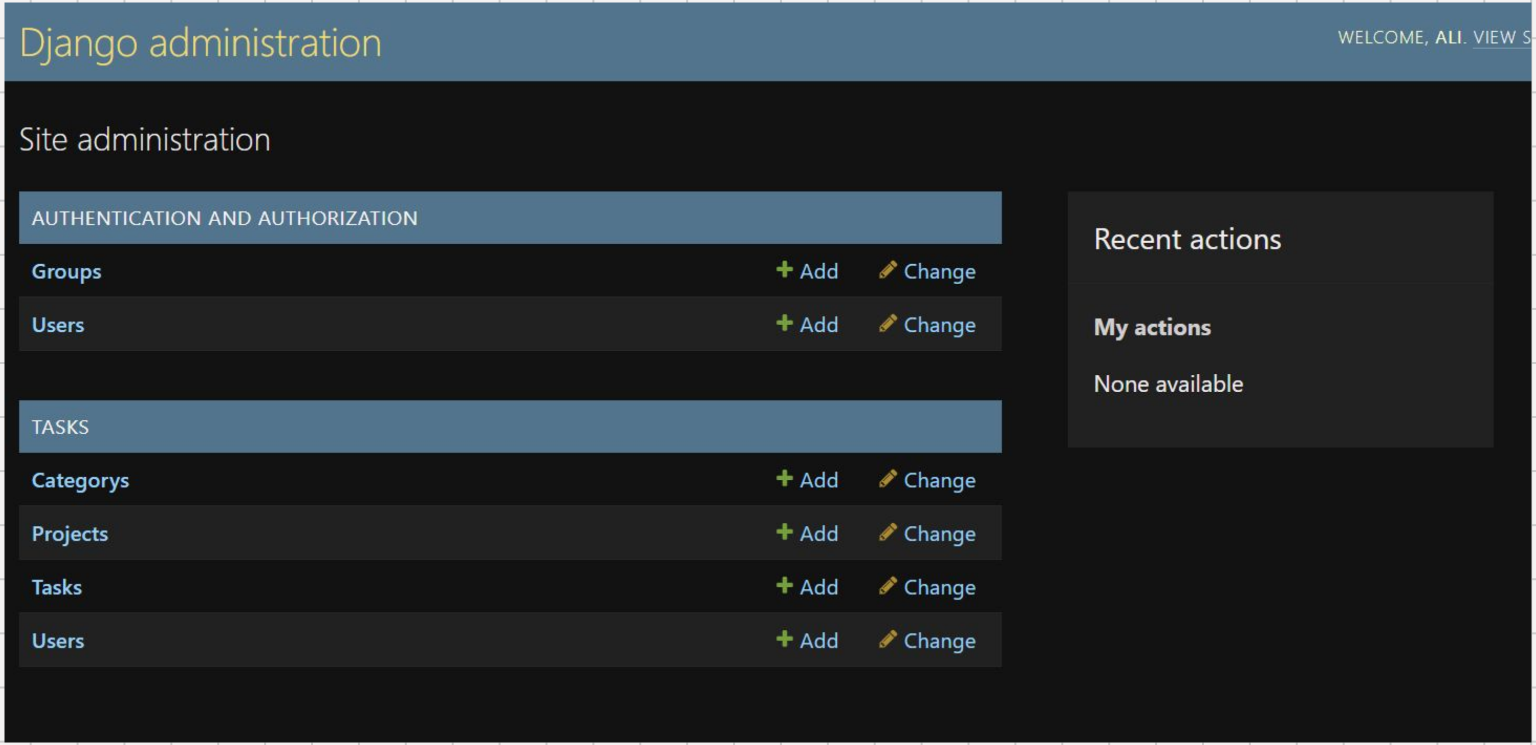


```
(venv) PS C:\Users\Ali\Desktop\Class Codes\Django Class Sessions> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
December 07, 2025 - 07:39:51
Django version 5.2.9, using settings 'taskmanager.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.
For more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/
```

Open Django Admin (Cont.)



Django administration

WELCOME, ALI. VIEW S

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

TASKS		
Categorys	+ Add	Change
Projects	+ Add	Change
Tasks	+ Add	Change
Users	+ Add	Change

Recent actions

My actions

None available

What Next?

The user needs a UI too!

How do we integrate that with a front-end?

Can Django render a front-end? If yes, how?