

# Model

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

# Model Class

Model class is a class which will represent a table in database.

Each model is a Python class that subclasses `django.db.models.Model`

Each attribute of the model represents a database field.

With all of this, Django gives you an automatically-generated database-access API

Django provides built-in database by default that is sqlite database.

We can use other database like MySQL, Oracle SQL etc.

# Create Our Own Model Class

models.py file which is inside application folder, is required to create our own model class.

Our own model class will inherit Python's Model Class.

Syntax:-


```
class ClassName(models.Model):  
    field_name=models.FieldType(arg, options)
```

Example:-

**models.py**

```
class Student(models.Model):  
    stuid=models.IntegerField()  
    stuname=models.CharField(max_length=70)  
    stuemail=models.EmailField(max_length=70)  
    stupass=models.CharField(max_length=70)
```

Length is required in CharField Type



- This class will create a table with columns and their data types
- Table Name will be ApplicationName\_ClassName, in this case it will be enroll\_student
- Field name will become table's Column Name, in this case it will be stuid, stuname, stuemail, stupass with their data type.
- As we have not mentioned primary key in any of these columns so it will automatically create a new column named 'id' Data Type Integer with primary key and auto increment.

# Create Our Own Model Class

## models.py

```
class Student(models.Model):
```

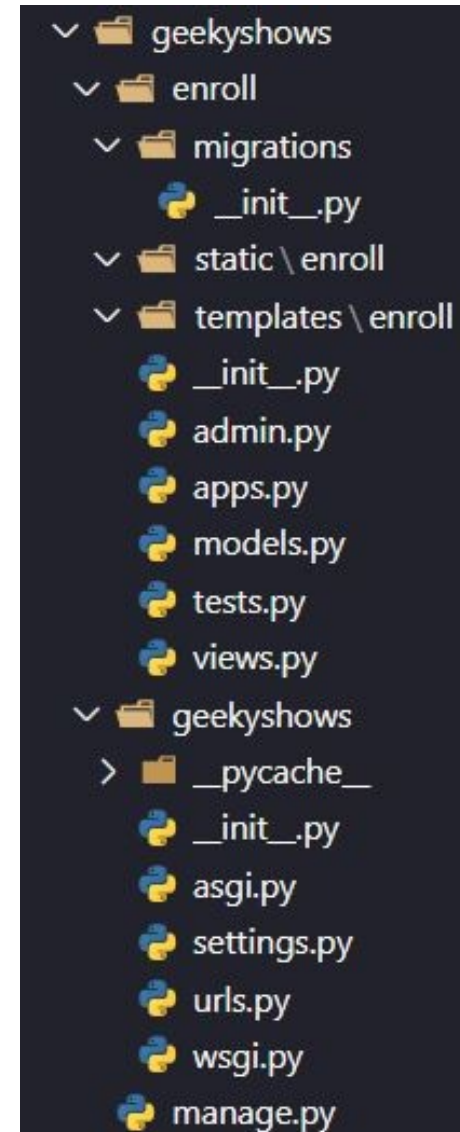
```
    stuid=models.IntegerField()
```

```
    stuname=models.CharField(max_length=70)
```

```
    stuemail=models.EmailField(max_length=70)
```

```
    stupass=models.CharField(max_length=70)
```

```
CREATE TABLE "enroll_student" (  
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    "stuid" integer NOT NULL,  
    "stuname" varchar(70) NOT NULL,  
    "stuemail" varchar(70) NOT NULL,  
    "stupass" varchar(70) NOT NULL  
);
```



# Rules

- Field Name instantiated as a class attribute and represents a particular table's Column name.
- Field Type is also known as Data Type.
- A field name cannot be a Python reserved word, because that would result in a Python syntax error.
- A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works.
- A field name cannot end with an underscore.

# How to use Models

Once you have defined your models, you need to tell Django you're going to use those models.

- Open settings.py file
- Write app name which contains models.py file in `INSTALLED_APPS = [ ]`
- Open Terminal
- Run `python manage.py makemigrations`
- Run `python manage.py migrate`

# Migrations

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema.

- **makemigrations** – This is responsible for creating new migrations based on the changes you have made to your models.
- **migrate** – This is responsible for applying and unapplying migrations.
- **sqlmigrate** – This displays the SQL statements for a migration.
- **showmigrations** – This lists a project's migrations and their status.

# makemigrations and migrate

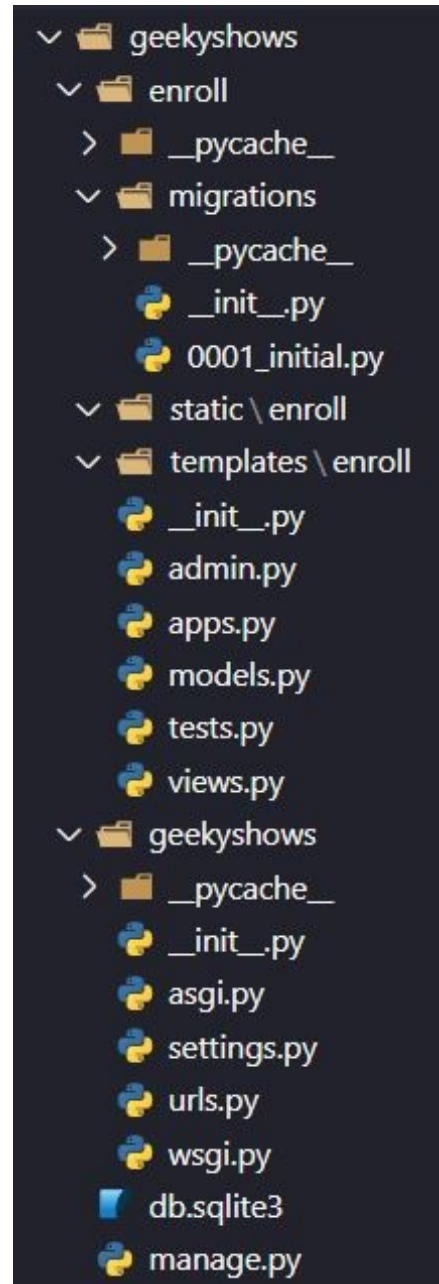
- **makemigrations** is used to convert model class into sql statements. This will also create a file which will contain sql statements. This file is located in Application's migrations folder.

Syntax:- `python manage.py makemigrations`

- **migrate** is used to execute sql statements generated by makemigrations. This command will execute All Application's (including built-in applications) SQL Statements if available. After execution of sql statements table will be created.

Syntax:- `python manage.py migrate`

Note – If you make any change in your own model class you are required to run makemigrations and migrate command only then you will get those changes in your application.





# Display SQL Statement

We can retrieve SQL Statement by using below command:-

Syntax:-

```
python manage.py sqlmigrate application_name dbfile_name
```

Example:-

```
python manage.py sqlmigrate enroll 0001
```

Note – File name can be found inside Application's migrations folder.

## **models.py**

```
class Student(models.Model):  
    stuid=models.IntegerField()  
    stuname=models.CharField(max_length=70)  
    stuemail=models.EmailField(max_length=70)  
    stupass=models.CharField(max_length=70)
```

## **python manage.py makemigrations**

```
CREATE TABLE "enroll_student" (  
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
    "stuid" integer NOT NULL,  
    "stuname" varchar(70) NOT NULL,  
    "stuemail" varchar(70) NOT NULL,  
    "stupass" varchar(70) NOT NULL  
);
```

## enroll/migrations/0001\_initial.py

```
from django.db import migrations, models
```

```
class Migration(migrations.Migration):
```

```
    initial = True
```

```
    dependencies = [ ]
```

```
    operations = [
```

```
        migrations.CreateModel(
```

```
            name='Student',
```

```
            fields=[
```

```
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
```

```
                ('stuid', models.IntegerField()),
```

```
                ('stuname', models.CharField(max_length=70)),
```

```
                ('stuemail', models.EmailField(max_length=70)),
```

```
                ('stupass', models.CharField(max_length=70)),
```

```
            ], ), ]
```

The “initial migrations” for an app are the migrations that create the first version of that app’s tables. Usually an app will have one initial migration, but in some cases of complex model interdependencies it may have two or more.

Initial migrations are marked with an `initial = True` class attribute on the migration class. If an initial class attribute isn’t found, a migration will be considered “initial” if it is the first migration in the app.

A list of migrations this one depends on

A list of Operation classes that define what this migration does

**enroll/migrations/0002\_student\_comment.py**

```
from django.db import migrations, models
```

```
class Migration(migrations.Migration):
```

```
    dependencies = [('enroll', '0001_initial'), ]
```

A list of migrations this one depends on

```
    operations = [
```

```
        migrations.AddField(
```

```
            model_name='student',
```

```
            name='comment',
```

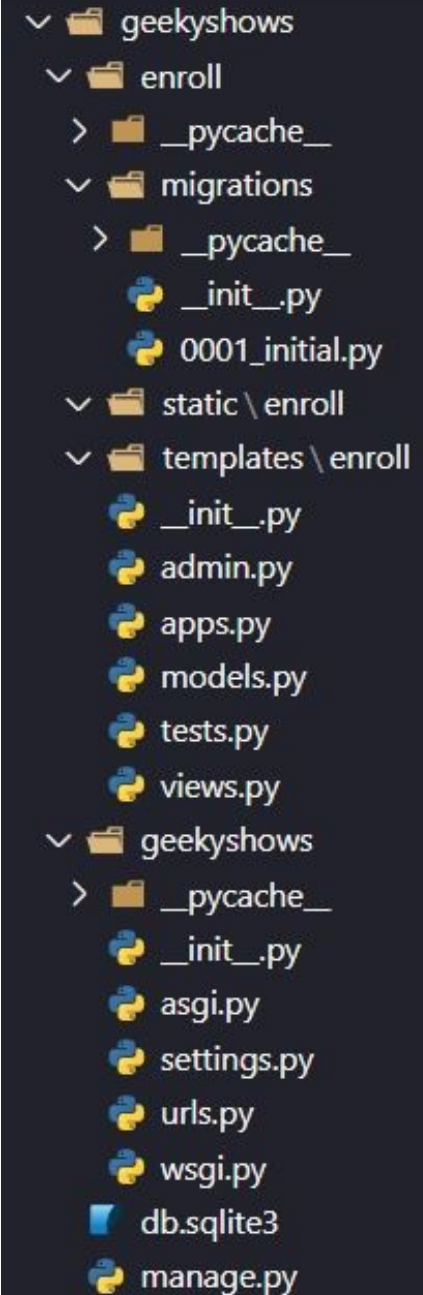
```
            field=models.CharField(default='not available', max_length=40),
```

```
        ), ]
```

A list of Operation classes that define what this migration does

# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application: *python manage.py startapp enroll*
- Add/Install Applications to Django Project using settings.py INSTALLED\_APPS
- Create **templates** folder inside application
- Create **static** folder inside application
- Open models.py file which is inside application
- Write Model Class
- Run *python manage.py makemigrations* Command
- A migration file will be generate automatically inside **migrations** folder
- Run *python manage.py migrate* Command
- Database table will be created automatically
- Write View Function inside views.py file
- Define url for view function of application using urls.py file
- Write Template files code
- Write Static file code



# Built-in Field Options

`null` – It can contain either `True` or `False`. If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

Avoid using `null` on string-based fields such as `CharField` and `TextField`. If a string-based field has `null=True`, that means it has two possible values for “no data”: `NULL`, and the empty string.

Example:- `null=True/False`

`blank` – It can contain either `True` or `False`. If `True`, the field is allowed to be blank. This is different than `null`. `null` is purely database-related, whereas `blank` is validation-related. If a field has `blank=True`, form validation will allow entry of an empty value. If a field has `blank=False`, the field will be required. Default is `False`.

Example:- `blank=True/False`

# Built-in Field Options

`default` - The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

Example:- `default='Not Available'`

`verbose_name` - A human-readable name for the field. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

Example:- `verbose_name='Student Name'`

`db_column` - The name of the database column to use for this field. If this isn't given, Django will use the field's name.

Example- `db_column = 'Student Name'`

# Built-in Field Options

`primary_key` – If True, this field is the primary key for the model.

If you don't specify `primary_key=True` for any field in your model, Django will automatically add an `AutoField` to hold the primary key, so you don't need to set `primary_key=True` on any of your fields unless you want to override the default primary-key behavior.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one. `primary_key=True` implies `null=False` and `unique=True`. Only one primary key is allowed on an object.

Example:- `primary_key=True`



# Built-in Field Options

unique - If True, this field must be unique throughout the table.

This is enforced at the database level and by model validation.

If you try to save a model with a duplicate value in a unique field, a `django.db.IntegrityError` will be raised by the model's `save()` method.

This option is valid on all field types except `ManyToManyField` and `OneToOneField`.

When `unique` is `True`, you don't need to specify `db_index`, because `unique` implies the creation of an index.

Example:- `unique=True`

# Built-in Field Types

IntegerField - An integer. Values from -2147483648 to 2147483647 are safe in all databases supported by Django. It uses MinValueValidator and MaxValueValidator to validate the input based on the values that the default database supports. The default form widget for this field is a NumberInput when localize is False or TextInput otherwise.

Example:- `roll = models.IntegerField()`

BigIntegerField - A 64-bit integer, much like an IntegerField except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807. The default form widget for this field is a TextInput.

Example:- `mobile = models.BigIntegerField()`

# Built-in Field Types

AutoField - An IntegerField that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify.

Example:- `stuid = models.AutoField()`

FloatField - A floating-point number represented in Python by a float instance. The default form widget for this field is a NumberInput when localize is False or TextInput otherwise.

Example:- `fees = models.FloatField()`

# Built-in Field Types

CharField - A string field, for small- to large-sized strings. For large amounts of text, use TextField. The default form widget for this field is a TextInput. CharField has one extra required argument: `max_length`

The maximum length (in characters) of the field.

Example:- `first_name = models.CharField(max_length=30)`

TextField - A large text field. The default form widget for this field is a Textarea. If you specify a `max_length` attribute, it will be reflected in the Textarea widget of the auto-generated form field. However it is not enforced at the model or database level. Use a CharField for that.

Example:- `description = models.TextField(max_length=150)`

# Built-in Field Types

BooleanField - A true/false field. The default form widget for this field is CheckboxInput, or NullBooleanSelect if null=True.

The default value of BooleanField is None when Field.default isn't defined.

Example:- `status = models.BooleanField()`

EmailField - A CharField that checks that the value is a valid email address using EmailValidator.

Example:- `email = models.EmailField()`

# Built-in Field Types

URLField - A CharField for a URL, validated by URLValidator. The default form widget for this field is a TextInput. Like all CharField subclasses, URLField takes the optional max\_length argument. If you don't specify max\_length, a default of 200 is used.

Example:- `partnersite = models.URLField()`

BinaryField - A field to store raw binary data. It can be assigned bytes, bytearray, or memoryview. By default, BinaryField sets editable to False, in which case it can't be included in a ModelForm. BinaryField has one extra optional argument: max\_length

The maximum length (in characters) of the field.

Example:- `profile_img = models.BinaryField()`

# Model Operations

CreateModel (name, fields, options=None, bases=None, managers=None) – It creates a new model in the project history and a corresponding table in the database to match it.

Where,

- name is the model name, as would be written in the models.py file.
- fields is a list of 2-tuples of (field\_name, field\_instance). The field instance should be an unbound field (so just models.CharField(...), rather than a field taken from another model).
- options is an optional dictionary of values from the model's Meta class.
- bases is an optional list of other classes to have this model inherit from; it can contain both class objects as well as strings in the format "appname.ModelName" if you want to depend on another model (so you inherit from the historical version). If it's not supplied, it defaults to inheriting from the standard models.Model.
- managers takes a list of 2-tuples of (manager\_name, manager\_instance). The first manager in the list will be the default manager for this model during migrations.

# Model Operations

DeleteModel(name) – It deletes the model from the project history and its table from the database.

RenameModel(old\_name, new\_name) – It renames the model from an old name to a new one.

You may have to manually add this if you change the model's name and quite a few of its fields at once to the autodetector, this will look like you deleted a model with the old name and added a new one with a different name, and the migration it creates will lose any data in the old table.

AlterModelTable(name, table) – It changes the model's table name (the db\_table option on the Meta subclass).

AlterUniqueTogether(name, unique\_together)- It changes the model's set of unique constraints (the unique\_together option on the Meta subclass).



# Model Operations

`AlterIndexTogether(name, index_together)` – It changes the model's set of custom indexes (the `index_together` option on the Meta subclass).

`AlterOrderWithRespectTo(name, order_with_respect_to)` – It makes or deletes the `_order` column needed for the `order_with_respect_to` option on the Meta subclass.

`AlterModelOptions(name, options)` – It stores changes to miscellaneous model options (settings on a model's Meta) like `permissions` and `verbose_name`. Does not affect the database, but persists these changes for RunPython instances to use. `options` should be a dictionary mapping option names to values.

`AlterModelManagers(name, managers)` – It alters the managers that are available during migrations.

# Model Operations

`AddField(model_name, name, field, preserve_default=True)` – It adds a field to a model.

Where

`model_name` is the model's name.

`name` is the field's name.

`field` is an unbound Field instance (the thing you would put in the field declaration in `models.py` for example, `models.IntegerField(null=True)`).

The `preserve_default` argument indicates whether the field's default value is permanent and should be baked into the project state (`True`), or if it is temporary and just for this migration (`False`) - usually because the migration is adding a non-nullable field to a table and needs a default value to put into existing rows. It does not affect the behavior of setting defaults in the database directly - Django never sets database defaults and always applies them in the Django ORM code.

# Model Operations

`RemoveField(model_name, name)` – It removes a field from a model.

Bear in mind that when reversed, this is actually adding a field to a model. The operation is reversible (apart from any data loss, which of course is irreversible) if the field is nullable or if it has a default value that can be used to populate the recreated column. If the field is not nullable and does not have a default value, the operation is irreversible.

`AlterField(model_name, name, field, preserve_default=True)` – It alters a field's definition, including changes to its type, null, unique, `db_column` and other field attributes.

The `preserve_default` argument indicates whether the field's default value is permanent and should be baked into the project state (True), or if it is temporary and just for this migration (False) - usually because the migration is altering a nullable field to a non-nullable one and needs a default value to put into existing rows. It does not affect the behavior of setting defaults in the database directly - Django never sets database defaults and always applies them in the Django ORM code.

Not all changes are possible on all databases - for example, you cannot change a text-type field like `models.TextField()` into a number-type field like `models.IntegerField()` on most databases.

# Model Operations

`RenameField(model_name, old_name, new_name)` – It changes a field's name (and, unless `db_column` is set, its column name).

`AddIndex(model_name, index)` – It creates an index in the database table for the model with `model_name`. `index` is an instance of the `Index` class.

`RemoveIndex(model_name, name)` – It removes the index named `name` from the model with `model_name`.

`AddConstraint(model_name, constraint)` – It creates a constraint in the database table for the model with `model_name`.

`RemoveConstraint(model_name, name)` – It removes the constraint named `name` from the model with `model_name`.