Assignment 4

1. HTML in Python:-

Introduction to embedding HTML within Python using webframeworks like Django or Flask:-

Embedding HTML within Python is a common practice when building web applications using web frameworks like **Django** or **Flask**. These frameworks allow you to dynamically generate web pages usi HTML templates combined with Python logic.

1. Flask

- Lightweight and beginner-friendly.
- You write Python code to define routes and logic.
- HTML is written in **templates** (usually using Jinja2 templating engine).

2. Django

- More full-featured than Flask.
- Includes admin panel, ORM, authentication, etc.
- Uses its own templating language to render HTML.

Example: Flask with Embedded HTML

app.py

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html", name="Alice")

if __name__ == "__main__":
    app.run(debug=True)
```

templates/index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

- {{ name }} is replaced by the value passed from Python (Alice).

Example: Django with Embedded HTML

views.py

```
from django.shortcuts import render

def home(request):
    return render(request, "index.html", {"name": "Alice"})
```

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

Generating dynamic HTML content using Django templates:-

Django templates are HTML files with special **template tags** and **variables** that allow you to dynamica
generate content from the backend.

Django View (views.py)

```
from django.shortcuts import render

def home(request):
    context = {
        "username": "Alice",
        "hobbies": ["Reading", "Gaming", "Hiking"]
    }
    return render(request, "home.html", context)
```

Here, the context dictionary sends data to the template.

Template (home.html)

html

CopyEdit

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome Page</title>
</head>
<body>
    <h1>Hello, {{ username }}!</h1>
```

```
    <h2>Your Hobbies:</h2>
    <ul>
        {% for hobby in hobbies %}
            <li>{{ hobby }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Output:

Hello, Alice!

Your Hobbies:
- Reading
- Gaming
- Hiking

2.CSS in Python:-

Integrating CSS with Django templates:-

To demonstrate how to generate dynamic HTML content using Django templates and how to integrate
for styling.

1. Django View (views.py)

```
from django.shortcuts import render

def home(request):
    context = {
        "username": "Alice",
        "hobbies": ["Reading", "Gaming", "Hiking"]
    }
    return render(request, "home.html", context)
```

2. Django Template (home.html)

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Welcome Page</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
</head>
<body>
    <h1>Hello, {{ username }}!</h1>
```

```
    <h2>Your Hobbies:</h2>
    <ul>
        {% for hobby in hobbies %}
            <li>{{ hobby }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

3. CSS File (style.css)

Location: myapp/static/myapp/style.css

```
body {
    background-color: #f5f5f5;
    font-family: Arial, sans-serif;
    padding: 20px;
}

h1 {
    color: #164e83;
    text-align: center;
}

ul {
    list-style-type: circle;
    padding-left: 30px;
}
```

4. File Structure

```
myproject/
 ├── myapp/
|    ├── views.py
|    ├── templates/
|    |   └── home.html
|    ├── static/
|    |   └── myapp/
|    |       └── style.css
```

5. settings.py (Static Files)

```
STATIC_URL = '/static/'
```

How to serve static files (like CSS, JavaScript) in Django:-

Serving static files (like CSS, JavaScript, and images) in Django is essential for styling and functionality your web app. Here's a simple, clear guide to help you do it properly.

1. Add Static File Settings in settings.py

```
# settings.py

# Static file URL prefix
STATIC_URL = '/static/'

# Optional (for custom directories)
STATICFILES_DIRS = [
    BASE_DIR / "static",  # if you have a global static folder
]
```

2. Create a Static Folder

You can store static files in either:

App-specific static files:

```
myapp/
 ├── static/
|   └── myapp/
|       ├── style.css
|       └── script.js
```

Project-level static files:

```
project_root/
 ├── static/
|    ├── style.css
|    └── images/
```

3. Use {% load static %} in Templates

At the top of your HTML template, load the static files:

```
{% load static %}
```

Then link the CSS/JS:

```
<link rel="stylesheet" href="{% static 'myapp/style.css' %}">
<script src="{% static 'myapp/script.js' %}"></script>
```

4. In Production (when DEBUG = False)

You need to collect static files into a single folder using:

```
python manage.py collectstatic
```

Make sure you've added this in settings.py:

```
STATIC_ROOT = BASE_DIR / "staticfiles"
```

Linking external or internal JavaScript files in Django.

1. Create Your JavaScript File

Inside your app's static folder:

```
myapp/
 ├── static/
|   └── myapp/
|       └── script.js
```

script.js Example

```
function showAlert() {
   alert("Hello from JavaScript!");
}

function toggleContent() {
   const content = document.getElementById("hidden-content");
   content.style.display = content.style.display === "none" ? "block" : "none";
}
```

2. Link JS in Your Template

home.html

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
   <title>Interactive Page</title>
   <script src="{% static 'myapp/script.js' %}" defer></script>
</head>
<body>
   <h1>JavaScript in Django Templates</h1>

   <button onclick="showAlert()">Click Me</button>

   <br><br>
   <button onclick="toggleContent()">Toggle Content</button>
   <div id="hidden-content" style="display: none;">
      <p>This is some toggleable content!</p>
   </div>
</body>
</html>
```

3. Make Sure Static Files Are Working

Make sure you've set this in settings.py:

```
STATIC_URL = '/static/'
```

1. Linking Internal JavaScript Files

Step 1: Create the JavaScript File

Put your JS file inside your app's static/ directory.

File structure:

```
myapp/
 ├── static/
|   └── myapp/
|       └── script.js
```

script.js example:

```
function greetUser() {
   alert("Hello from internal JavaScript!");
}
```

Step 2: Link It in Your Template

Use {% load static %} at the top and link like this:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
   <title>My Page</title>
   <script src="{% static 'myapp/script.js' %}" defer></script>
</head>
<body>
   <button onclick="greetUser()">Greet Me</button>
</body>
</html>
```

Use defer to make sure JS loads after the HTML.

2. Linking External JavaScript Libraries

You can link libraries like jQuery, Bootstrap, or CDN-hosted scripts by placing their URLs directly in the template.

Example: Link jQuery via CDN

```
<!DOCTYPE html>
<html>
<head>
   <title>Using jQuery</title>
   <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>
   <button id="show-msg">Click me</button>
   <p id="message" style="display:none;">Hello from jQuery!</p>
```

```
    <script>
        $(document).ready(function(){
            $("#show-msg").click(function(){
                $("#message").show();
            });
        });
    </script>
</body>
</html>
```

4. Django Introduction:-

Overview of Django: Web development framework:-

Django is a high-level Python web framework that promotes rapid development and clean, pragmatic design. It was created to help developers build secure, maintainable websites quickly.

Key Features

MVC Architecture (called MVT in Django):
Model – handles database.
View – handles logic.
Template – handles UI.
Batteries-Included: Comes with everything you need (ORM, authentication, admin panel, forms, etc.).
Security: Protects against common threats like SQL injection, CSRF, XSS, and more.
Scalability: Suitable for both small projects and large-scale enterprise applications.
DRY Principle: "Don't Repeat Yourself" – encourages reusability and efficiency.

Components

Models: Define your database schema using Python classes.
Views: Control the logic and return HTTP responses.
Templates: HTML files with Django Template Language (DTL) for dynamic content.
Admin Interface: Auto-generated backend for managing data.
URL Dispatcher: Clean, readable URLs with routing via urls.py.

Use Cases

Content Management Systems (CMS)
E-commerce platforms
Social networks
Scientific computing platforms
APIs using Django REST Framework (DRF)

Advantages of Django (e.g., scalability, security).

1. **Security**
    - Automatically protects against common web threats like:
        - **SQL injection**
        - **Cross-Site Scripting (XSS)**
        - **Cross-Site Request Forgery (CSRF)**
        - **Clickjacking**
    - Comes with built-in authentication and user management.
2. **Rapid Development**
    - Lets you build web apps **quickly** with reusable components.
    - Reduces development time with built-in features like forms, admin interface, and ORM.
3. **DRY Principle (Don't Repeat Yourself)**
    - Encourages reusable code and clean architecture.
    - Helps maintain consistency and reduces redundancy.
4. **Scalability**
    - Used by high-traffic sites like Instagram and Pinterest.
    - Can handle millions of users with proper optimization.
5. **Batteries-Included**
    - Comes with a wide range of built-in tools:
        - Admin panel
        - ORM (Object-Relational Mapper)
        - URL routing
        - Form handling
        - Middleware support
6. **Versatile & Full-Stack**
    - Great for both frontend and backend.
    - You can build everything from APIs to e-commerce sites.
7. **Robust ORM**
    - Maps Python classes to database tables.
    - Makes database queries easy, readable, and secure.
8. **Admin Interface**
    - Auto-generated and customizable admin dashboard for managing database models.
9. **Excellent Documentation & Community**
    - Comprehensive official docs.
    - Large community = lots of tutorials, plugins, and support.
10. **Easy Integration**
    - Works well with third-party tools and libraries.
    - Can integrate with frontend frameworks (like React or Vue) and REST APIs.

Django vs. Flask comparison: Which to choose and why.

Django

- Full-stack **web framework** (batteries-included).
- Comes with built-in **admin panel**, **authentication**, and **ORM**.
- Follows **convention over configuration** (less flexibility, more structure).
- Ideal for **large projects** and **rapid development**.
- More secure out of the box (CSRF, SQL injection protection).
- Has a steep learning curve but great for production-level apps.
- Example use case: **E-commerce sites**, **social media platforms**, **CMS**.

Flask

- Lightweight **micro-framework**.
- Minimal setup, gives **more control and flexibility**.
- Doesn't come with built-in tools — you choose what to add (e.g., ORM, forms).
- Great for **smaller projects** or building **REST APIs**.
- Easier to learn and customize.
- Ideal when you want to keep the app **light and modular**.
- Example use case: **Microservices**, **simple web apps**, **APIs**.

Which to Choose?

- Choose **Django** if:
  - You want a ready-made structure and tools.
  - You're building something big and production-ready.
  - You prefer fast development with built-in features.
- Choose **Flask** if:
  - You want full control over your stack.
  - You're building a small app or REST API.
  - You prefer minimalism and adding only what you need.

5. Virtual Environment:-

A **virtual environment** is an isolated environment that allows you to manage dependencies separately each Python project. It creates a self-contained directory with its own Python interpreter and libraries, independent from the system-wide Python installation.

Importance of a Virtual Environment

1. **Avoid Dependency Conflicts**
   - Different projects might need different versions of the same package.
   - Virtual environments prevent conflicts by keeping dependencies isolated.
2. **Clean and Organized Projects**
   - All project-specific packages are kept inside the virtual environment folder.
   - Keeps your global Python installation clean.
3. **Reproducibility**
   - You can create a requirements.txt file that lists all packages used.
   - Others can recreate the same environment using pip install -r requirements.txt.

4. **Project-Specific Python Versions**
    - Use different Python versions for different projects if needed.
5. **Safe Testing**
    - Safely test new packages or upgrades without affecting other projects or the system.

Using venv or virtualenv to create isolated environments.

Using venv (Built-in in Python 3.3+)

venv is a built-in module in Python 3, used to create virtual environments.

Steps:

1. **Create a virtual environment:**

```
python -m venv env
```

This creates a folder named env containing the isolated Python environment.

2. **Activate the virtual environment:**
    - **On Windows:**

```
env\Scripts\activate
```

3. **Deactivate the environment:**

```
Deactivate
```

6. Project and App Creation:-

Steps to create a Django project and individual apps within the project.

Steps to Create a Django Project & Apps

1. Install Django

Make sure you're in a virtual environment first:

```
pip install django
```

2. Create a Django Project

django-admin startproject myproject
cd myproject
This creates a new folder structure like:

```
myproject/
├── manage.py
└── myproject/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── asgi.py
    └── wsgi.py
```

✅ **3. Run the Development Server**

python manage.py runserver

Open http://127.0.0.1:8000 to see the default Django welcome page.

✅ 4. Create an App Within the Project

python manage.py startapp myapp

This creates:

```
myapp/
 ├── admin.py
 ├── apps.py
 ├── models.py
 ├── views.py
 ├── urls.py (create this manually)
 ├── tests.py
└── migrations/
```

5. Add the App to Project Settings

Open myproject/settings.py and add your app to INSTALLED_APPS:

```
INSTALLED_APPS = [
    ...
    'myapp',
]
```

Understanding the role of manage.py, urls.py, and views.py.

manage.py

- It's the **command-line utility** for managing your Django project.
- Acts as a **wrapper** around Django's administrative commands.
- You use it to:
  - Run the development server (python manage.py runserver)
  - Apply migrations (python manage.py migrate)
  - Create apps (python manage.py startapp appname)
  - Create superusers, manage database, etc.

Think of it as: the main control center for your Django project.

urls.py

- Responsible for **routing URLs** to the appropriate views.
- You define **URL patterns** that determine what view is shown for which URL.
- Can be split into **project-level urls.py** and **app-level urls.py**.

Think of it as: the traffic controller of your website.

Example:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
]
```

views.py

- Contains the **logic** for what should happen when a user visits a specific URL.
- It returns a response, often an HTML page or JSON data.
- Views can be simple functions or class-based.

Think of it as: the brain that processes requests and returns responses.

Example:

```
from django.http import HttpResponse

def home(request):
   return HttpResponse("Welcome to my site!")
```

7. MVT Pattern Architecture:-

Django's MVT Architecture

Django follows the **Model-View-Template (MVT)** architectural pattern, which is similar to the Model-V Controller (MVC) pattern. It separates the application into three interconnected components, helping developers organize code and manage complex web applications efficiently.

1. Model

- The **Model** is the data access layer.
- It handles all interactions with the database.
- Each model is a Python class that maps to a table in the database.
- It defines the structure of the stored data, including fields, data types, and relationship

2. View

- The **View** is the business logic layer.
- It receives HTTP requests, processes data (often by interacting with the model), and returns an HTTP response.
- Views decide **what data to display** and **how to process user requests**, but not how to present it—that's the job of templates.

3. Template

- The **Template** is the presentation layer.
- It defines the structure and layout of the final HTML page using the Django Template Language (DTL).
- Templates are HTML files with placeholders for dynamic content, passed from views.

Request-Response Cycle in Django

1. **User sends a request**: When a user types a URL or clicks a link, the browser sends an H request to the Django server.
2. **URL Dispatcher (urls.py)**: Django's URLconf system matches the incoming request URL defined URL pattern and routes it to the corresponding view function.
3. **View Processes Request**:
    - The view receives the request.
    - It interacts with the **Model** to retrieve or manipulate data as needed.
    - It then passes this data to the **Template** for rendering.

4. **Template Renders Data**:
    - The **Template** receives the data and dynamically generates an HTML page.
    - It presents the data in a user-friendly format using the Django template engine.
5. **Response Sent to User**:
    - The final rendered HTML is returned as an **HTTP response** to the user's browser
    - The browser displays the content to the user.

8. Django Admin Panel:-

Django comes with a **powerful built-in admin interface**, which is one of its standout features. The Djan admin panel provides a **ready-to-use interface** for managing application data. It is **automatically gener** from the models you define, and it enables developers and administrators to **perform CRUD operation** (Create, Read, Update, Delete) on database records without writing any extra code.

Key Features of Django Admin Panel

- **Auto-generated interface** based on your models.
- Supports **authentication and permissions** (only authorized users can access it).
- Allows **adding, editing, and deleting records**.
- Easily customizable to fit project-specific needs.
- Integrates well with Django's user model and groups.
- Helps in rapid development and testing of data models.

Enabling and Accessing the Admin Panel

1. **Ensure 'django.contrib.admin' is in INSTALLED_APPS** (default setting).
2. **Run migrations** to set up the necessary database tables:

    python manage.py migrate
3. **Create a superuser** to access the admin panel:

    python manage.py createsuperuser

    python manage.py runserver
4. **Login to the admin panel** at:

    http://127.0.0.1:8000/admin/

Registering Models in Admin

To make your models appear in the admin interface, you need to register them in your app's admin.py file:

```
from django.contrib import admin
from .models import Product

admin.site.register(Product)
```

Customizing the Django admin interface to manage database records.

1. Registering Models with Custom Admin Classes

Instead of registering a model directly, you can use a custom admin class to control how the model app in the admin panel.

```
from django.contrib import admin
from .models import Product

class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'available', 'created_at')  # Fields shown in the list view
    search_fields = ('name',)  # Adds a search box
    list_filter = ('available', 'category')  # Adds sidebar filters
    ordering = ('-created_at',)  # Default sorting order
    fields = ('name', 'description', 'price', 'available')  # Fields to show in the form

admin.site.register(Product, ProductAdmin)
```

2. Customizing Field Display

- **list_display** – shows selected fields as columns in the object list.
- **list_filter** – adds filters in the right sidebar.
- **search_fields** – enables a search bar for specified fields.
- **ordering** – sets the default sorting order.
- **fields / fieldsets** – controls which fields appear in the form and their layout.

3. Inline Models (Related Data Editing)

You can manage related models directly from the parent model page using **inline editing**.

```
class OrderItemInline(admin.TabularInline):
    model = OrderItem
    extra = 1  # Number of extra empty forms

class OrderAdmin(admin.ModelAdmin):
    inlines = [OrderItemInline]

admin.site.register(Order, OrderAdmin)
```

4. Custom Form Widgets and Validation

You can override the default form used in the admin to add custom validation or widgets.

```
from django import forms
from .models import Product

class ProductForm(forms.ModelForm):
    class Meta:
        model = Product
        fields = '__all__'
        widgets = {
            'description': forms.Textarea(attrs={'rows': 4, 'cols': 40}),
```

```
    }

class ProductAdmin(admin.ModelAdmin):
    form = ProductForm

admin.site.register(Product, ProductAdmin)
```

5. Restricting Access and Permissions

You can control who can view or edit what using Django's **permissions system** or override admin meth
like:

```
def has_change_permission(self, request, obj=None):
    return request.user.is_superuser  # Only superusers can edit
```

9. URL Patterns and Template Integration:-

Setting up URL patterns in urls.py for routing requests to views.

Setting up URL patterns in urls.py is a key part of building a Django application, as it allows you to route
incoming HTTP requests to the appropriate views.

Here's a basic example of how to set up URL patterns in urls.py:

Project Structure Example

```
myproject/
 ├── myapp/
|    ├── views.py
|    ├── urls.py   ← app-specific urls
 ├── myproject/
|    ├── urls.py   ← main project urls
```

1. In myapp/urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),  # home page
    path('about/', views.about, name='about'),  # about page
]
```

2. In myproject/urls.py:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),  # include the app's urls
]
```

Integrating templates with views to render dynamic HTML content.

Integrating templates with views to render dynamic HTML content is a common task in web developm
Here's a concise guide to help you understand the process using a popular web framework like Django:

1. Setting Up Your Template
First, create an HTML template. This template will contain placeholders for dynamic content.
Example: templates/my_template.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ heading }}</h1>
    <p>{{ content }}</p>
</body>
</html>
```

2. Creating a View
Next, create a view that will render this template and pass dynamic data to it.
Example: views.py

```
from django.shortcuts import render

def my_view(request):
    context = {
        'title': 'Dynamic Page Title',
        'heading': 'Welcome to My Page',
        'content': 'This is a dynamically generated content.'
    }
    return render(request, 'my_template.html', context)
```

3. Configuring URLs
Ensure that your view is accessible by configuring the URL routing.
Example: urls.py

```
from django.urls import path
from .views import my_view

urlpatterns = [
    path('my-page/', my_view, name='my_page'),
]
```

4. Running the Server

Finally, run your Django development server to see the dynamic content in action.

python manage.py runserver

Summary

- **Template**: Contains placeholders for dynamic content.
- **View**: Passes data to the template.
- **URL Configuration**: Maps the view to a URL.

10. Form Validation using JavaScript:-

Using JavaScript for front-end form validation.

JavaScript form validation checks user input in the browser before sending data to the server, improvin user experience and reducing unnecessary server requests. It ensures required fields are filled correctl helps catch simple input errors early.

HTML + Simple JavaScript Validation

```
<!DOCTYPE html>
<html>
<head>
  <title>Simple Form Validation</title>
</head>
<body>

  <h2>Signup Form</h2>
  <form onsubmit="return validateForm()">
    <label>Username:</label>
    <input type="text" id="username"><br><br>

    <label>Email:</label>
    <input type="text" id="email"><br><br>

    <input type="submit" value="Submit">
  </form>

  <script>
    function validateForm() {
      var username = document.getElementById("username").value;
      var email = document.getElementById("email").value;

      if (username === "") {
        alert("Username is required!");
        return false;
      }
```

```
    if (email === "") {
      alert("Email is required!");
      return false;
    }

    // basic email pattern
    if (!email.includes("@") || !email.includes(".")) {
      alert("Please enter a valid email address.");
      return false;
    }

    return true; // allow form to submit
  }
 </script>

</body>
</html>
```

11. Django Database Connectivity (MySQL or SQLite):-

Connecting Django to a database (SQLite or MySQL):-

1. Default: Using SQLite (No Setup Needed)

SQLite is the default database used when you create a new Django project. It's file-based and works ou
the box.

Check your settings.py:

```
# settings.py

DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.sqlite3',
      'NAME': BASE_DIR / 'db.sqlite3',
   }
}
```

2. Using MySQL

Step 1: Install MySQL Client

```
pip install pymysql
```

Step 2: Update settings.py

```
# settings.py
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'your_db_name',
        'USER': 'your_db_user',
        'PASSWORD': 'your_db_password',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Using the Django ORM for database queries.

Using the Django ORM
Django's ORM (Object-Relational Mapper) allows developers to interact with the database using Pytho
code instead of writing SQL queries. It translates Python classes and methods into SQL queries behind
scenes.

- Key Features:

Model-based: You define models (Python classes) that represent database tables.
Abstraction: You can create, retrieve, update, and delete records using simple Python methods.
Security: Automatically handles SQL injection protection and sanitization.
Database-agnostic: Works with multiple database backends (SQLite, MySQL, PostgreSQL, etc.) without
changing the ORM code.

- Core Operations:

Create: Model.objects.create() or instance.save()
Read: Model.objects.all(), filter(), get(), exclude()
Update: Modify attributes and call .save()
Delete: Use .delete() on a model instance
Advanced Queries: Use lookups like __icontains, __gte, __lte, and chaining for filtering

12. ORM and QuerySets:-

Understanding Django's ORM and how QuerySets are used to interact with the database.

What is Django's ORM?
Django's **Object-Relational Mapper (ORM)** allows you to interact with your database using Python obj
instead of writing SQL manually.
When you define a model in Django, it:

- Creates a table in your database.
- Provides a Python interface to create, retrieve, update, and delete records (CRUD operations).

Defining a Model

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    enrolled = models.BooleanField(default=True)
```

Behind the scenes, this creates a table like:

```
CREATE TABLE student (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(100),
    age INTEGER,
    enrolled BOOLEAN
);
```

What is a QuerySet?

A **QuerySet** is a Django object that represents a collection of database rows.

Key Features of QuerySets:

- **Lazy Evaluation**: QuerySets are not run until needed.
- **Chaining**: You can chain QuerySet methods to build complex queries.

Basic QuerySet Examples

1. **Get All Records**

```
Student.objects.all()
```

2. **Filter Records**

```
Student.objects.filter(age__gte=18)
```

3. **Get One Record**

```
Student.objects.get(id=1)
```

4. **Create a Record**

```
Student.objects.create(name="Alice", age=20)
```

5. **Update Records**

```
Student.objects.filter(name="Alice").update(age=21)
```

6. Delete Records

Student.objects.filter(name="Alice").delete()

13. Django Forms and Authentication:-

sing Django's built-in form handling

Using Django's built-in form handling involves creating a form class, rendering the form in a template, processing form submissions securely in a view. Django provides two main classes for form handling:

1. forms.Form

Used when you want to create a custom form not tied to a database model.

- **Define the form**: You declare each field explicitly.
- **Render the form**: Use a Django template to render the form.
- **Validate and process**: In the view, check if the form is valid, then use the cleaned data

2. forms.ModelForm

Used when the form is based on a model. It automatically generates form fields based on the model's fields.

- **Define the model and form**: Django will map model fields to form fields.
- **On submission**: You can create or update model instances using the form data.

Features of Django Form Handling:

- Automatic form field generation.
- Built-in validation rules.
- Clean separation of concerns between views, templates, and data models.
- Secure by default: includes CSRF protection and data sanitization.

Implementing Django's authentication system (sign up, login, logout, password management).

Django provides a complete, secure, and customizable **authentication system** out of the box, which handles **sign up**, **login**, **logout**, and **password management** (reset, change, etc.).

Key Components of Django's Authentication System:

1. User Model

Django includes a built-in User model with fields like username, email, password, etc.
You can import it using:

from django.contrib.auth.models import User

Functionality Overview

1. User Signup (Registration)

You can use a UserCreationForm to register new users.

```
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render, redirect

def signup_view(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('login')  # Redirect to login after signup
    else:
        form = UserCreationForm()
    return render(request, 'signup.html', {'form': form})
```

2. User Login

Use Django's built-in AuthenticationForm and login() function.

```
from django.contrib.auth.forms import AuthenticationForm
from django.contrib.auth import login, authenticate

def login_view(request):
    if request.method == 'POST':
        form = AuthenticationForm(data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            return redirect('home')  # Redirect to home page
    else:
        form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})
```

3. User Logout

Use the logout() function.

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    return redirect('login')  # Redirect after logout
```

4. Password Management

a. Change Password (Logged-in user)

Use PasswordChangeForm.

b. Reset Password (Forgot Password)

Use Django's built-in views and templates:

- PasswordResetView
- PasswordResetDoneView
- PasswordResetConfirmView
- PasswordResetCompleteView

Configure email backend and URLs for reset links.

URLs Setup

Add these to urls.py to use Django's built-in auth views:

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', login_view, name='login'),
    path('logout/', logout_view, name='logout'),
    path('signup/', signup_view, name='signup'),

    # Password management
    path('password_change/', auth_views.PasswordChangeView.as_view(), name='password_change'),
    path('password_reset/', auth_views.PasswordResetView.as_view(), name='password_reset'),
]
```

Templates

Django expects templates like registration/login.html, registration/password_reset_form.html, etc., or can render your own with custom paths.

14. CRUD Operations using AJAX:-

Using AJAX for making asynchronous requests to the server without reloading the page.

Using **AJAX** in Django allows you to send asynchronous requests to the server without reloading the pa enabling dynamic content updates and a smoother user experience.

AJAX (Asynchronous JavaScript and XML) allows web pages to retrieve data from the server and update parts of the page without a full reload. In Django, this is typically done using JavaScript (often jQuery) t send a request to the server, where a view processes the request and returns a response (usually in JSO format). The response is then used by the front-end JavaScript to update the page content.

AJAX Workflow in Django:

1. **Frontend (JavaScript)**: JavaScript (or jQuery) sends an asynchronous request to the Dja server, specifying the URL and method (GET or POST).
2. **Backend (Django View)**: Django processes the request in a view and returns a respons often using JsonResponse to send data in JSON format.
3. **Frontend (JavaScript)**: The data is then processed by JavaScript on the client side, whic updates the web page dynamically without refreshing.

AJAX can be used for various purposes:

- **Form submission**: Send form data without reloading the page.
- **Dynamic content**: Load additional content (e.g., more items on a list) or update conten dynamically.
- **Real-time updates**: Fetch data like notifications, messages, or live search results witho reloading the page.

AJAX in Django is an efficient way to create dynamic and interactive web applications, enhancing the us experience while reducing server load and page reloads.

15. Customizing the Django Admin Panel:-

Techniques for customizing the Django admin panel.

Customizing the Django admin panel allows you to tailor the administrative interface to better suit the needs of your project and users. Django's admin is highly customizable, enabling you to adjust the layo behavior, and functionality to make it more intuitive and efficient for managing data.

1. Customizing Admin Model Display

The simplest way to customize the admin panel is by modifying how models are displayed in the admin interface.

a. Using list_display

You can customize which fields are displayed in the list view for a model.

```
from django.contrib import admin
from .models import Product

class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'stock', 'category')

admin.site.register(Product, ProductAdmin)
```

This will display name, price, stock, and category columns in the list view of the Product model.

b. Adding Filters

You can add filters on the right side of the admin interface to allow users to filter by specific fields.

```
class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'category')
    list_filter = ('category',)

admin.site.register(Product, ProductAdmin)
```

c. Adding Search Functionality

Enable search functionality for a model based on specific fields.

```
class ProductAdmin(admin.ModelAdmin):
    search_fields = ['name', 'category']

admin.site.register(Product, ProductAdmin)
```

2. Customizing Forms

You can customize the forms used in the admin panel by overriding the form attribute in the admin cla
This allows you to use custom forms for the model.

```
from django import forms
from django.contrib import admin
from .models import Product

class ProductForm(forms.ModelForm):
    class Meta:
        model = Product
        fields = ['name', 'price', 'description']

class ProductAdmin(admin.ModelAdmin):
    form = ProductForm

admin.site.register(Product, ProductAdmin)
```

3. Inline Models

If a model has a relationship with another model, you can use **inline models** to display related models
the same form.

```
from django.contrib import admin
from .models import Category, Product

class ProductInline(admin.TabularInline):  # Or admin.StackedInline for a different layout
    model = Product
    extra = 1  # Number of empty forms to display initially

class CategoryAdmin(admin.ModelAdmin):
    inlines = [ProductInline]
```

```
admin.site.register(Category, CategoryAdmin)
```

4. Customizing Admin Actions

You can define custom actions to perform bulk operations on selected items in the admin panel.

```
from django.contrib import admin
from .models import Product

def mark_as_in_stock(modeladmin, request, queryset):
    queryset.update(stock=True)

mark_as_in_stock.short_description = 'Mark selected products as in stock'

class ProductAdmin(admin.ModelAdmin):
    actions = [mark_as_in_stock]

admin.site.register(Product, ProductAdmin)
```

5. Custom Admin Views

Sometimes, you may need custom views or additional information in the admin panel. Django provides
way to add custom views to the admin interface.

```
from django.urls import path
from django.http import HttpResponse
from django.contrib import admin

class MyAdminSite(admin.AdminSite):
    def get_urls(self):
        urls = super().get_urls()
        custom_urls = [
            path('custom-view/', self.custom_view)
        ]
        return custom_urls + urls

    def custom_view(self, request):
        return HttpResponse("This is a custom view!")

admin_site = MyAdminSite()
admin.site = admin_site  # Replace default admin site

# Register models for custom admin site
admin_site.register(Product)
```

6. Customizing the Admin Interface with JavaScript and CSS

Django allows you to include custom JavaScript and CSS in the admin panel.

- To include custom CSS or JavaScript, you can override the Media class in your ModelAd
  class.

```
class ProductAdmin(admin.ModelAdmin):
    class Media:
        css = {
            'all': ('myapp/css/admin.css',)
        }
        js = ('myapp/js/admin.js',)

admin.site.register(Product, ProductAdmin)
```

7. Customizing the Admin Form Layout

You can customize the layout of the fields in the form by adjusting the fieldsets attribute.

```
class ProductAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('name', 'price')
        }),
        ('Advanced options', {
            'classes': ('collapse',),
            'fields': ('description', 'category', 'stock'),
        }),
    )

admin.site.register(Product, ProductAdmin)
```

8. Customizing Admin Permissions

You can control who has access to specific actions and data in the admin panel by using Django's built-i
permissions system.

- For example, you can restrict access to certain fields based on the user's permissions.

```
class ProductAdmin(admin.ModelAdmin):
    def get_readonly_fields(self, request, obj=None):
        if not request.user.has_perm('myapp.change_product'):
            return ['price']
        return super().get_readonly_fields(request, obj)

admin.site.register(Product, ProductAdmin)
```

9. Adding Dashboard Widgets

For complex sites, you may want to display important information on the admin dashboard (e.g., sales
reports, product statistics). This can be done by overriding the index view and adding custom template

16. Payment Integration Using Paytm:-

Introduction to integrating payment gateways (like Paytm) in Django projects.

Integrating a **payment gateway** like **Paytm** in a Django project allows you to accept online payments securely. Payment gateways handle the transaction processing, user authentication, and communicatio with banks or wallets.

Introduction to Payment Gateway Integration in Django

1. What is a Payment Gateway?

A payment gateway is a service that authorizes and processes payments (via credit/debit cards, UPI, wallets, etc.) between your website and the customer's bank.

Steps to Integrate Paytm in Django:

1. Create a Paytm Merchant Account

- Sign up at https://business.paytm.com
- Get your **Merchant ID (MID)**, **Merchant Key**, and **Website name** (from the dashboard)

2. Install Required Libraries

Install packages like requests (if needed for server-to-server communication).

```
pip install requests
```

3. Set Up Payment URLs and Keys in Django Settings

Add Paytm credentials in settings.py:

```
PAYTM_MERCHANT_KEY = 'your_merchant_key'
PAYTM_MERCHANT_ID = 'your_merchant_id'
PAYTM_WEBSITE = 'WEBSTAGING'  # Use 'DEFAULT' or as given for production
PAYTM_CALLBACK_URL = 'http://yourdomain.com/handle_payment_response/'
```

4. Create Views for Payment Processing

- Generate checksum using Paytm's utility.
- Redirect the user to Paytm with transaction details.

Example View:

```
from django.shortcuts import render
from django.conf import settings
from .PaytmChecksum import generateChecksum  # You can get this from Paytm Docs

def initiate_payment(request):
    data = {
        'MID': settings.PAYTM_MERCHANT_ID,
        'ORDER_ID': 'ORDER0001',
        'CUST_ID': 'cust001',
```

```
        'TXN_AMOUNT': '500.00',
        'CHANNEL_ID': 'WEB',
        'WEBSITE': settings.PAYTM_WEBSITE,
        'INDUSTRY_TYPE_ID': 'Retail',
        'CALLBACK_URL': settings.PAYTM_CALLBACK_URL,
    }
    data['CHECKSUMHASH'] = generateChecksum(data, settings.PAYTM_MERCHANT_KEY)
    return render(request, 'redirect_to_paytm.html', {'paytm_data': data})
```

5. Create HTML Form to Redirect to Paytm

```
<form method="post" action="https://securegw-stage.paytm.in/order/process">
  {% for key, value in paytm_data.items %}
    <input type="hidden" name="{{ key }}" value="{{ value }}">
  {% endfor %}
  <input type="submit" value="Pay Now">
</form>
```

6. Handle Payment Response

Paytm sends a POST request to your callback URL with transaction data. Validate the checksum and up
payment status.

17. GitHub Project Deployment:-

Steps to push a Django project to GitHub.

1. Create a GitHub Repository

- Go to https://github.com
- Click **New Repository**
- Give it a **name**, add a **README** if you like (optional), and click **Create repository**

2. Initialize Git in Your Django Project

In your terminal, navigate to your Django project folder:

```
cd path/to/your/project
git init
```

3. Create a .gitignore File

Add a .gitignore file to exclude unnecessary files (e.g., migrations, secrets, etc.). You can use this sampl

```
*.pyc
__pycache__/
db.sqlite3
/static/
media/
```

.env
*.log
Optional: Use django.gitignore from GitHub's gitignore templates.

4. Add and Commit Your Files
git add .
git commit -m "Initial commit"

5. Link to Your GitHub Repository
Replace the URL with your actual GitHub repo URL:
git remote add origin https://github.com/your-username/your-repo-name.git

6. Push to GitHub
git branch -M main
git push -u origin main

Optional but Recommended:
Never push sensitive data!
Use an .env file for secrets (e.g., secret keys, database passwords) and keep it excluded in .gitignore.

18. Live Project Deployment (PythonAnywhere):-

Introduction to deploying Django projects to live servers like PythonAnywhere.

Deploying a **Django project** to a live server like **PythonAnywhere** allows you to make your application accessible on the internet. PythonAnywhere is a beginner-friendly, cloud-based platform specifically designed for hosting Python applications, including Django.

Introduction to Deploying Django on PythonAnywhere:
Here are the basic steps:
1.**Sign Up and Log In to PythonAnywhere**
Visit https://www.pythonanywhere.com and create an account.
2.**Upload Your Django Project**
You can upload your code via the **"Files"** tab or push it to GitHub and **clone** it on PythonAnywhe using:

git clone https://github.com/yourusername/yourproject.git

1.**Set Up a Virtual Environment (optional but recommended**
Create and activate a virtual environment in the **Bash console**:

python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

1. **Configure the Web App**
   - Go to the **"Web"** tab and click **"Add a new web app"**.
   - Choose **"Manual configuration"**, select **Python version**, and enter the path to your Django wsgi. file.
2. **Set Environment Variable**
   - Under the **"Web" → "Environment"**, set DJANGO_SETTINGS_MODULE to your project's settings (e.g., myproject.settings).
3. **Database Migration**
   - Run the following in the **Bash console**:

python manage.py migrate

1. **Collect Static Files**
   - Configure STATIC_ROOT in settings.py, then run:

python manage.py collectstatic

2. **Reload the Web App**
   - Hit the **"Reload"** button from the **"Web" tab** to apply changes and start the app

19. Social Authentication:-

Here's a concise explanation of how to **set up social login (Google, Facebook, GitHub)** in Django using **OAuth2**, commonly via the **django-allauth** package:

Steps to Set Up Social Login in Django (Google, Facebook, GitHub):

Install Required Packages

pip install django-allauth

1. **Add to INSTALLED_APPS in settings.py**

```
INSTALLED_APPS = [
    ...
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',
    'allauth.socialaccount.providers.github',
]
SITE_ID = 1
```

2. **Update AUTHENTICATION_BACKENDS**

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
)
```

3. **Include Allauth URLs**

   In urls.py:

```
path('accounts/', include('allauth.urls')),
```

4. **Configure Redirect URLs**

   In settings.py:

```
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
```

5. **Register Your App with the OAuth Providers**
   - **Google**: https://console.developers.google.com
   - **Facebook**: https://developers.facebook.com
   - **GitHub**: https://github.com/settings/developers

   Get your **Client ID** and **Secret Key**.

6. **Add Provider Credentials in Django Admin**

   Go to **Social Applications** in Django admin and add credentials for each provider:
   - Name, Client ID, Secret Key, and the sites (usually example.com or localhost).

7. **Run Migrations**

```
python manage.py migrate
```

20. Google Maps API:-

Integrating Google Maps API into Django projects.

Steps to Integrate Google Maps API in Django:

1. **Get a Google Maps API Key**
   - Go to https://console.cloud.google.com
   - Create a project and enable the **Maps JavaScript API**
   - Generate an **API key**
2. **Include the Maps API Script in Your Template**

   Add this in your HTML template (usually inside <head> or before </body>):

```
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap" async defer></script>
```

3. **Add a Map Container**

   In your HTML template:

```
<div id="map" style="height: 400px; width: 100%;"></div>
```

4. **Initialize the Map with JavaScript**

Add a script block to your template:

```
<script>
 function initMap() {
  const location = { lat: 28.6139, lng: 77.2090 };  // Example: Delhi
  const map = new google.maps.Map(document.getElementById("map"), {
   zoom: 10,
   center: location,
  });
  const marker = new google.maps.Marker({
   position: location,
   map: map,
  });
 }
</script>
```

5. **Optional: Pass Dynamic Coordinates from Django View**

You can send coordinates via context:

```
def show_map(request):
    context = {
        'lat': 28.6139,
        'lng': 77.2090,
    }
    return render(request, 'map.html', context)
```

And use them in JavaScript:

```
const location = { lat: {{ lat }}, lng: {{ lng }} };
```