❖ HTML in Python

1. Introduction to embedding HTML within Python using web frameworks like Django or Flask

❖ Embedding HTML within Python using web frameworks like **Django** or **Flask** refers to the process of generating dynamic web pages by combining Python backend logic with HTML templates. These frameworks use template engines to pass data from Python code to HTML files, ensuring a clear separation between application logic and presentation. This approach improves code organization, maintainability, and allows the development of interactive and scalable web applications.

2. Generating dynamic HTML content using Django templates.

❖ Generating dynamic HTML content using **Django templates** allows web pages to display content that changes according to user input or data from the server. In Django, the view handles the application logic and sends required data to the template. The template then uses placeholders and template tags to insert this data into the HTML structure.

Django templates support features such as variable display, conditional statements, and loops, which help in showing dynamic content like lists, user information, or messages. This method follows the principle of separation of concerns, where Python manages logic and HTML handles presentation, making the application easy to maintain and secure.

Lab:

Write a Python program to render an HTML file using Django's template system.

Step 1: Create a Django project and app

```
pip install virtualenv
 python -m venv myenv
 cd myenv
 scripts\activate
 pip install django
 django-admin startproject myproject
 cd myproject
 django-admin startapp myapp
 python manage.py migrate
 python manage.py runserver
```

Step 2: Add app to settings.py

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

Step 3: Create a template folder

myapp/templates /index.html

Step 4: Create HTML file(index.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Finder</title>
</head>
<body>
  <h1>Welcome to Doctor Finder</h1>
</body>
</html>
```

Step 5: Create View (views.py)

```
from django.shortcuts import render
def index(request):
  return render(request, "index.html")
```

Step 6: URL Configuration(urls.py)

```
from django.urls import path
from myapp import views
urlpatterns = [
  path('', views.index, name='index'),
]
```

Step 7: Connect app URLs to project

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
   path('admin/', admin.site.urls),
   path('', include('myapp.urls')),
]
```

Step 8: Run the server

```
python manage.py runserver
```

❖ CSS in Python

1. **Integrating CSS with Django templates**

- Integrating CSS with Django templates is essential for improving the visual appearance and user interface of a web application. Django uses a static files system to manage CSS, JavaScript, and images separately from HTML templates. CSS files are stored in a dedicated static directory and linked to HTML templates using template tags.
- By connecting CSS files to Django templates, developers can apply consistent styling across multiple web pages. This approach supports reusability, as the same CSS file can be used by different templates. Integrating CSS with Django templates also follows the principle of separation of concerns, where

design and styling are kept separate from application logic, resulting in cleaner, more maintainable web applications.

2.How to serve static files (like CSS, JavaScript) in Django.

- In Django, static files such as CSS, JavaScript, and images are managed using the static files framework. These files are kept separate from templates and Python code to maintain a clean project structure.
  Loading Static Files in Templates
  Django templates use a special template tag to access static files.

```
{% load static %}
<link rel="stylesheet" href="{% static 'css/style.css' %}">
```

Lab:

Create a CSS file to style a basic HTML template in Django.

Step 1: Create a css file to style a basic HTML template in Django
myapp/static/css/style.css
Step 2: CSS File (style.css)

```
body {
  background-color: #f2f2f2;
  font-family: Arial;
}
h1 {
  color: #2c3e50;
  text-align: center;
}
```

Step 3: Load Static Files in HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Profile</title>
  <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
  <h1>Doctor Profile</h1>
</body>
</html>
```

Django project to display a webpage with custom CSS styling for a doctor profile page

(doctor.html)

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Profile</title>
  <link rel="stylesheet" href="{% static 'doctor.css' %}">
</head>
```

```
<body>
<div class="profile">
  <h1>Dr. Anjali Sharma</h1>
  <p>Specialization: Pediatrician</p>
  <p>Experience: 8 Years</p>
  <p>Hospital: City Care Hospital</p>
</div>
</body>
</html>
```

(views.py)

```
from django.shortcuts import render
def doctor_profile(request):
  return render(request, "doctor.html")
```

❖ JavaScript with Python

1. Using JavaScript for client-side interactivity in Django templates.

- JavaScript is used in Django templates to add client-side interactivity such as button clicks, form validation, and dynamic content updates. Django renders HTML on the server, while JavaScript runs in the browser to handle user actions. JavaScript can be included using <script> tags or external static files. Django template tags allow passing data safely from the backend to JavaScript.

- **Example:**

```
<button id="btn">Click</button>
<script>
 document.getElementById("btn").addEventListener("click",
function () {
   alert("Hello from Django!");
 });
</script>
```

2. **Linking external or internal JavaScript files in Django.**
   In Django, JavaScript files are linked using the static files system. External JavaScript files are stored inside the static folder of an app and loaded in templates using the {% static %} template tag. Internal JavaScript can also be written directly inside the HTML template using <script> tags. Using external JavaScript files is recommended for better code organization and reusability.
   Example (External JS):

```
{% load static %}
<script src="{% static 'app/main.js' %}"></script>
```

Lab :

Django project with JavaScript-enabled form validation

Step 1: Create Django Project & App

django-admin startproject hospital

cd hospital

python manage.py startapp patient

Step 2: Register App in settings.py

```
INSTALLED_APPS = [
    'patient',
]
```

Step 3: Create View (views.py)

```
from django.shortcuts import render
def register(request):
    return render(request, "register.html")
```

Step 4: URL Configuration(urls.py)

```
from django.urls import path
from patient import views
urlpatterns = [
    path('', views.register, name='register'),
]
```

hospital/urls.py

```
from django.urls import path, include
urlpatterns = [
    path('', include('patient.urls')),
]
```

Step 5: JavaScript File

static/js/validate.js

```
function validateForm() {
    let name = document.forms["regForm"]["name"].value;
    let age = document.forms["regForm"]["age"].value;

    if (name === "" || age === "") {
        alert("All fields are required");
        return false;
    }
    if (age < 0) {
        alert("Invalid age");
        return false;
    }
    return true;
}
```

Step 6: HTML Template

templates/register.html

```
{% load static %}
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Patient Registration</title>
  <script src="{% static 'js/validate.js' %}"></script>
</head>
<body>
  <h2>Patient Registration</h2>
  <form name="regForm" onsubmit="return validateForm()">
    Name: <input type="text" name="name"><br><br>
    Age: <input type="number" name="age"><br><br>
    <input type="submit" value="Register">
  </form>
</body>
</html>
```

❖ Django Introduction

1. Overview of Django: Web development framework

Django is a high-level Python web development framework used to build secure and scalable web applications quickly. It follows the Model–View–Template (MVT) architecture pattern. Django provides built-in features like authentication, URL routing, ORM, and admin panel. It emphasizes rapid development and clean, reusable code. Django is widely used for developing dynamic and database-driven websites.

2. **Advantages of Django (e.g.,scalability, security).**
   Django provides high security by protecting applications from common attacks like SQL injection, XSS, and CSRF. It is highly scalable and can handle large traffic with proper configuration. Django enables rapid development through its built-in features and reusable components. It includes an automatic admin interface that saves development time. Django also supports clean and maintainable code using its MVT architecture.

3. **Django vs. Flask comparison: Which to choose and why.**
   Django is a full-featured web framework that provides built-in tools like authentication, admin panel, and ORM, making it suitable for large and complex applications. Flask is a lightweight and flexible framework that gives developers more control and is ideal for small or simple projects. Django follows a structured approach, while Flask allows more customization. Choose Django for scalable, secure, and database-driven applications, and Flask for lightweight apps or learning purposes.

   Lab: •
   Write a short project using Django's built-in tools to render a simple webpage

View (views.py)

```
from django.shortcuts import render

def index(request):
    return render(request, "index.html")
```

Template (index.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>Django Page</title>
</head>
<body>
  <h1>Hello from Django!</h1>
</body>
</html>
```

❖ Virtual Environment

1. **Understanding the importance of a virtual environment in Python projects**

   A virtual environment is an isolated Python setup created for a specific project. It allows developers to install and manage project-specific packages without affecting the global Python installation. This helps avoid dependency conflicts between different projects. Virtual environments also make projects easier to maintain and deploy. Overall, they ensure a clean, stable, and reproducible development environment.

2. **Using venv or virtualenv to create isolated environments.**

   venv and virtualenv are tools used to create isolated Python environments for projects. They allow each project to have its own

Python interpreter and dependencies, preventing version conflicts. The environment can be activated before installing packages so they remain project-specific. This helps in maintaining clean and consistent development setups.

Example:

```
python -m venv myenv
myenv\Scripts\activate
virtualenv myenv
```

Lab:

Set up a virtual environment for a Django project.

pip install virtualenv

python -m venv myenv

cd myenv

scripts\activate

pip install django

django-admin startproject myproject

cd myproject

django-admin startapp myapp

python manage.py migrate

python manage.py runserver

❖ Project and App Creation

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

   First, install Django and create a new project using the django-admin startproject command. Next, move into the project directory and run the development server to check setup. To create an app, use the python manage.py startapp command. Finally, register the app in INSTALLED_APPS inside settings.py.

   Example:

```
django-admin startproject myproject

cd myproject

python manage.py startapp myapp

python manage.py runserver
```

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

- manage.py is a command-line utility used to run and manage Django project tasks like starting the server and creating apps. urls.py is responsible for mapping URLs to specific views in the application. views.py contains the logic that handles user requests and returns responses such as web pages or JSON data. Together, these files control how requests are processed in a Django project.

Lab:

Create a Django project with an app to manage doctor profiles.

Step 1: Create Django Project

django-admin startproject myproject

cd myproject

Step 2: Create App

python manage.py startapp myapp

Step 3: Register App in settings.py

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

Step 4: Create View (doctor/views.py)

```
from django.shortcuts import render
def doctor(request):
   return render(request, "doctor.html")
```

Step 5: URL Configuration

urls.py

```
from django.urls import path
from myapp import views
urlpatterns = [
   path('doctor/', views.doctor , name=’doctor’),
]
```

Step 6: Create Template

templates/doctor.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Profile</title>
</head>
<body>
  <h1>Doctor Profile Page</h1>
</body>
</html>
```

❖ MVT pattern Architecture

1. **Django's MVT (Model-View-Template) architecture and how it handles request-response cycles.**

   Django uses the Model–View–Template (MVT) architecture to organize web applications. The Model manages the database and handles data operations. The View processes user requests and contains application logic. The Template is responsible for displaying data as HTML to the user. When a request is made, Django maps the URL to a view. The view interacts with the model and renders a template. The final response is then sent back to the user's browser.

Lab:

Build a simple Django app showcasing how the MVT architecture works.

MVT Architecture

- Model – handles data
- View – handles logic
- Template – handles presentation

Model (M)

models.py

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=50)
    course = models.CharField(max_length=50)
    def __str__(self):
        return self.name
```

View (V)

views.py

```
from django.shortcuts import render
from .models import Student


def student_view(request):
```

```
    student = Student(name="Niyati", course="Information Technology")
    return render(request, 'student.html', {'student': student})
```

Template (T)

templates/student.html

```
<!DOCTYPE html>
<html>
<head>
    <title>MVT Example</title>
</head>
<body>
    <h1>Student Details</h1>
    <p>Name: {{ student.name }}</p>
    <p>Course: {{ student.course }}</p>
</body>
</html>
```

❖ Django Admin Panel

1. Introduction to Django's built-in admin panel.

- Django provides a powerful built-in admin panel that allows developers to manage application data easily. It is automatically generated and requires minimal configuration. The admin panel

supports creating, updating, deleting, and viewing database records. It includes authentication and permission management by default. Models can be registered to appear in the admin interface. This feature saves development time and simplifies backend management.

2. **Customizing the Django admin interface to manage database records.**

- Django allows customization of the admin interface to manage database records efficiently. Developers can modify how models are displayed by registering them with custom admin classes. Fields can be listed, searched, filtered, and ordered in the admin panel. Forms and layouts can also be customized for better usability. Permissions can be controlled for different users. This makes the admin panel more user-friendly and powerful for data management.

Lab:
Set up and customize the Django admin panel to manage a "Doctor Finder" project

Step 1: Create Superuser

python manage.py createsuperuser

Step 2: Register Model in Admin (doctor/admin.py)

from django.contrib import admin

from .models import Doctor

```
class DoctorAdmin(admin.ModelAdmin):
    list_display = ('name', 'specialization', 'experience')
    search_fields = ('name', 'specialization')
admin.site.register(Doctor, DoctorAdmin)
```

Step 3: Run Server

```
python manage.py runserver
```

Step 4: Access Admin Panel

http://127.0.0.1:8000/admin/

❖ URL Patterns and Template Integration

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

- In Django, urls.py is used to define URL patterns for routing requests to views. Each URL pattern maps a specific URL path to a corresponding view function or class. Django checks the URL patterns in order and finds the first match. The path() or re_path() function is used to define routes. Named URLs help in referencing paths easily within templates. This system controls how user requests reach the correct views.

2.Integrating templates with views to render dynamic HTML content.

- In Django, templates are used to generate dynamic HTML pages. Views collect data from models or other sources and pass it to

templates as context. The render() function is used to combine a template with context data. Templates use Django template language to display variables and logic. This separation keeps business logic and presentation separate. As a result, dynamic and reusable web pages are created efficiently.

Lab:

Create a Django project with URL patterns and corresponding views and templates.

Step 1: Create Project and App

```
django-admin startproject doctorfinder

cd doctorfinder

python manage.py startapp doctor
```

Step 2: Register App in settings.py

```
INSTALLED_APPS = [
    'doctor',
]
```

Step 3: Create Views (doctor/views.py)

```
from django.shortcuts import render


def home(request):
    return render(request, "home.html")
```

```
def profile(request):
    return render(request, "profile.html")

def contact(request):
    return render(request, "contact.html")
```

Step 4: URL Configuration

doctor/urls.py

```
from django.urls import path
from myapp import views
urlpatterns = [
    path('', views.home, name='home'),
    path('profile/', views.profile, name='profile'),
    path('contact/', views.contact, name='contact'),
]
```

Step 5: Templates

templates/home.html

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Doctor Finder</title>
</head>
<body>
    <h1>Welcome to Doctor Finder</h1>
    <a href="/profile/">Profile</a> |
    <a href="/contact/">Contact</a>
</body>
</html>
```

templates/profile.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Doctor Profile</title>
</head>
<body>
    <h2>Dr. John Smith</h2>
    <p>Specialization: Cardiologist</p>
    <a href="/">Home</a>
</body>
```

```
</html>
```

templates/contact.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Contact</title>
</head>
<body>
  <h2>Contact Us</h2>
  <p>Email: contact@doctorfinder.com</p>
  <a href="/">Home</a>
</body>
</html>
```

❖ Form validation using JavaScripts

1. Using JavaScript for front-end form validation.

JavaScript is used for front-end form validation to check user input before submitting the form. It helps ensure fields like name, email, and password are filled correctly. Validation reduces server load by catching errors early. JavaScript can display instant error messages to users. Event listeners are used to validate forms on submit. This improves user experience and data accuracy.

Lab:

Write a Django project to implement JavaScript form validation for a user registration form.

Registration Form with JavaScript Validation

templates/register.html

```
<!DOCTYPE html>
<html>
<head>
  <title>User Registration</title>
  <script>
    function validateForm() {
      let name = document.forms["regForm"]["username"].value;
      let email = document.forms["regForm"]["email"].value;
      let password = document.forms["regForm"]["password"].value;
      if (name == "") {
        alert("Username is required");
        return false;
      }
      if (email == "") {
        alert("Email is required");
        return false;
      }
```

```
        if (password.length < 6) {
            alert("Password must be at least 6 characters");
            return false;
        }
        return true;
    }
  </script>
</head>
<body>
<h2>User Registration</h2>
<form name="regForm" onsubmit="return validateForm()">
  Username: <input type="text" name="username"><br><br>
  Email: <input type="email" name="email"><br><br>
  Password: <input type="password" name="password"><br><br>
  <input type="submit" value="Register">
</form>
</body>
</html>
```

❖ Django Database Connectivity(MySQL)

1. **Connecting Django to a database (SQLite or MySQL).**
   Django can be connected to a MySQL database using PyMySQL as the database driver. First, PyMySQL is installed using pip. Then, the database configuration is updated in the DATABASES section of settings.py with MySQL credentials. PyMySQL is added as the MySQL client so Django can communicate with the database. After configuration, Django migrations are run to create database tables. This setup allows Django to perform database operations using its ORM.

2. **Using the Django ORM for database queries.**

- Django ORM (Object Relational Mapper) allows developers to interact with the database using Python code instead of writing SQL queries. It maps database tables to Python classes called models. Using ORM, data can be created, retrieved, updated, and deleted easily. Common methods like objects.create(), objects.filter(), and objects.get() are used for queries. The ORM improves code readability and database security. It also supports multiple databases without changing query syntax.

  Lab:

  Set up database connectivity for a Django project

  Step 1: Install PyMySQL

  pip install pymysql

  step 2: Create Database in MySQL

  import pymysql

  db_name = "my_django_db"

```
connection = pymysql.connect(
  host="localhost",
  user="root",
  password=""
)
cursor = connection.cursor()
cursor.execute(f"CREATE DATABASE IF NOT EXISTS {db_name}")
print("Database connectivity established successfully")
connection.close()
```

❖ ORM and QuerySets

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

Django's ORM (Object Relational Mapper) provides an easy way to interact with the database using Python code. It represents database tables as models and rows as objects. A QuerySet is a collection of database queries that retrieves objects from the database. QuerySets are lazy, meaning queries are executed only when data is needed. They allow filtering, ordering, and slicing of data. This makes database interaction simple, efficient, and database-independent.

Lab:

Perform CRUD operations using Django ORM

Create (Insert Data)

```
Doctor.objects.create(
   name="Dr. Alice",
   specialization="Dermatologist",
   experience=8
)
```

Read (Fetch Data)

```
doctors = Doctor.objects.all()

for doc in doctors:
   print(doc.name, doc.specialization, doc.experience)
```

Update (Modify Data)

```
doc = Doctor.objects.get(id=1)
doc.experience = 10
doc.save()
```

Delete (Remove Data)

```
doc = Doctor.objects.get(id=1)

doc.delete()
```

❖ Django Forms and Authentication

1. **Using Django's built-in form handling.**
   Django provides a built-in form handling system to manage user input easily and securely. Forms are created using Django's forms module and are linked to HTML forms. They automatically handle data validation and error messages. Django forms protect against security issues like CSRF attacks. Cleaned data can be accessed after validation. This system reduces manual work and improves code reliability.

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

   - The signup function handles user registration by checking if the email already exists in the database. If the passwords match, a new user is created and stored in the database along with profile image and user type. If the email already exists or passwords do not match, an error message is shown.
   - The login function authenticates users by matching email and password from the database. On successful login, user details like email and profile are stored in the session. Based on the user type (buyer or seller), the user is redirected to different pages. If login fails, appropriate error messages are displayed.
   - The logout function clears the session data and redirects the user to the login page. This ends the user session securely.

Lab:

Create a Django project for user registration and login functionality.

```
def signup(request):
    form = SignupForm(request.POST or None)
    if form.is_valid():
        user = form.save(commit=False)
        user.set_password(form.cleaned_data['password'])
        user.save()
        return redirect('login')
    return render(request, "signup.html", {'form': form})


def user_login(request):
    if request.method == "POST":
        user = authenticate(
            username=request.POST['username'],
            password=request.POST['password']
        )
        if user:
            login(request, user)
            return redirect('profile')
    return render(request, "login.html")
```

```
def user_logout(request):
    logout(request)
    return redirect('login')
```

❖ CRUD Operations using AJAX

1. **Using AJAX for making asynchronous requests to the server without reloading the page.**
   AJAX is used to send and receive data from the server without reloading the web page. It allows the browser to communicate with Django views asynchronously. Using AJAX improves user experience by updating only specific parts of the page. JavaScript sends requests using fetch() or XMLHttpRequest. Django processes the request and returns data, usually in JSON format. This technique is commonly used for forms, search, and live updates.

   Lab:
   Implement AJAX in a Django project for performing CRUD operations.

Step 1: View (doctor/views.py)

```
from django.http import JsonResponse
from .models import Doctor
def add_doctor(request):
    if request.method == "POST":
        Doctor.objects.create(
```

```
        name=request.POST['name'],
        specialization=request.POST['specialization']
    )
    return JsonResponse({'status': 'success'})
```

Step 2: URL Configuration

```
path('add/', add_doctor, name='add_doctor'),
```

Step 3: Template with AJAX

templates/doctor.html

```
<script>
function addDoctor() {
  fetch("/add/", {
    method: "POST",
    headers: {
      "X-CSRFToken": "{{ csrf_token }}"
    },
    body: new URLSearchParams({
      name: document.getElementById("name").value,
      specialization: document.getElementById("spec").value
    })
```

```
  })
  .then(res => res.json())
  .then(data => alert("Doctor added successfully"));
}
</script>

<input type="text" id="name" placeholder="Name">
<input type="text" id="spec" placeholder="Specialization">
<button onclick="addDoctor()">Add Doctor</button>
```

❖ Customizing the Django Admin Panel

1. **Techniques for customizing the Django admin panel.**

    - Register models in admin.py to display them in the admin panel
    - Use list_display to show specific fields in the list view
    - Use search_fields to enable searching of records
    - Use list_filter to filter data easily
    - Customize form layout and fields
    - Control user permissions and access levels

    Lab:
    Customize the Django admin panel for better management of records

Steps:

1. Create a Django project and app.
2. Define a Doctor model with fields like name, specialty, availability, city, and phone in models.py.
3. Register the Doctor model in admin.py using a custom DoctorAdmin class that sets list_display, search_fields, and
4. Customize site branding using admin.site.site_header, etc., in admin.py to match the “Doctor Finder” theme.
5. Run makemigrations, migrate, create a superuser, and log in to /admin/ to verify the customized interface.

❖ Payment Integration Using Paytm

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

   - Payment gateways are used to handle online payments securely
   - Razorpay can be integrated with Django for processing payments
   - API keys provided by Razorpay are used for authentication
   - Django views create and manage payment orders
   - JavaScript handles payment checkout on the frontend
   - Razorpay verifies payment status after completion
   - This integration enables safe and reliable online transactions

   Lab:

   Implement Paytm payment gateway in a Django project.

   In a Django project, Paytm payment gateway integration generally involves the following steps:

1. **Merchant Registration**
   The developer first registers on the Paytm Merchant platform to obtain required credentials such as Merchant ID and Merchant Key.
2. **Payment Request Creation**
   When a user places an order, Django prepares payment details like order ID, amount, and customer information and sends them to Paytm.
3. **Redirect to Paytm Page**
   The user is redirected to the Paytm payment page where the payment is completed securely.
4. **Payment Processing**
   Paytm processes the payment and verifies the transaction.
5. **Response Handling**
   After payment, Paytm sends a response back to the Django application indicating success or failure of the transaction.
6. **Order Status Update**
   Based on the response, Django updates the payment status in the database and displays the result to the user.

❖ Live Project Deployment(PythonAnywhere)

1. **Introduction to deploying Django projects to live servers like PythonAnywhere.**

   - Deployment means making a Django project accessible on the internet
   - PythonAnywhere is a cloud platform used to host Django applications
   - The Django project is uploaded to the server
   - Virtual environment is created on PythonAnywhere
   - Project dependencies are installed using pip
   - Web app and WSGI file are configured
   - After setup, the Django site becomes live for users

Lab:

Deploy a Django project to PythonAnywhere.

PythonAnywhere is an online cloud platform that allows users to host Python and Django applications.
Deploying a Django project on PythonAnywhere makes the application available on the internet without using a local server.

To deploy a Django project on PythonAnywhere:

1. Create an account on PythonAnywhere.
2. Upload the Django project files to PythonAnywhere.
3. Set up a virtual environment and install Django.
4. Configure a new web application by selecting Django framework.
5. Update Django project settings for deployment.
6. Reload the web application to make it live.

After successful deployment, the Django project can be accessed using the URL provided by PythonAnywhere.

❖ Social Authentication

1. **Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.**

    - Social login allows users to sign in using Google, Facebook, or GitHub accounts
    - Django uses OAuth2 protocol for secure third-party authentication
    - django-allauth is commonly used to implement social login

- OAuth credentials (Client ID and Secret) are obtained from providers
- Providers are configured in Django settings.py
- URLs and templates are set up for login buttons
- This feature improves user convenience and authentication security

Lab:

Implement Google and Facebook login for the Django project.

Google and Facebook login provide social authentication, which allows users to log in to a Django application using their existing Google or Facebook accounts.
This method improves user experience by eliminating the need to create a separate username and password.

In a Django project, social login integration generally follows these steps:

1. Create Developer Accounts
   The developer registers the application on Google Developer Console and Facebook Developer Portal to obtain client credentials.
2. Configure Authentication Settings
   Django authentication settings are configured to support third-party login providers such as Google and Facebook.
3. User Authentication Flow
   When a user clicks on Google or Facebook login, they are redirected to the respective platform for authentication.
4. Permission and Verification
   The user grants permission to share basic profile information like name and email.
5. Callback and Login
   After successful verification, the user is redirected back to the Django application and logged in automatically.

6. Account Creation or Linking
   If the user is new, a new account is created. If the user already exists, the account is linked and logged in.

Social login ensures secure authentication using OAuth technology.

❖ Google Maps API

1. Integrating Google Maps API into Django projects.

- Google Maps API is used to display maps in Django web applications
- An API key is generated from the Google Cloud Console
- The API key is added to the Django template
- JavaScript is used to load and display the map
- Locations can be shown using markers on the map
- Django views pass location data to templates
- This integration helps in location-based services and navigation

Lab:

Use Google Maps API to display doctor locations in the "Doctor Finder" project.

In the *Doctor Finder* project, the Google Maps API is used to display the geographical locations of doctors on an interactive map.

Each doctor's location is stored in the system using address or latitude and longitude values.

To implement this, a Google Maps API key is generated from the Google Cloud Console and integrated into the Django project.

The map is displayed on a web page, and doctor locations are marked using map markers.

When a user opens the Doctor Finder page, the map loads and shows the locations of available doctors.
This helps users easily find nearby doctors and view their locations visually.