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

- Python is widely used for web development. It cannot show web pages directly, so web frameworks like Django and Flask are used. These frameworks help connect Python code with HTML pages to create dynamic websites.

Embedding HTML in Python

Embedding HTML in Python means displaying data from Python inside HTML pages using templates. Python handles logic, and HTML handles design.

Django and HTML

Django uses a template system to send data from Python to HTML.

Example:

```
return render(request, "home.html", {"name": "User"})
<h1>Welcome {{ name }}</h1>
```

Flask and HTML

Flask uses Jinja2 templates to embed Python data in HTML.

Example:

```
return render_template("index.html", name="User")
<h2>Hello {{ name }}</h2>
```

2. Generating dynamic HTML content using Django templates.

- Django is a popular Python web framework used to build dynamic websites. It allows developers to generate dynamic HTML content using its built-in **template system**. Templates help display data from Python into HTML pages.

What is Dynamic HTML?

Dynamic HTML means web pages where content changes based on:

- User input
- Database data
- Conditions or logic

Django templates make this possible by combining HTML with Python data.

Django Template System

Django uses **Django Template Language (DTL)**, which supports:

- Variables
- Loops
- Conditions
- Template inheritance

Passing Data from View to Template

Python views send data to templates using a dictionary.

View (views.py):

```
from django.shortcuts import render

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

Displaying Data in HTML

Use double curly braces to show data.

Template (home.html):

```
<h1>Welcome {{ name }}</h1>
```

3. Integrating CSS with Django templates.

- CSS (Cascading Style Sheets) is used to design and style web pages. In Django, CSS files are connected to HTML templates using **static files**. This allows developers to separate design from logic and create attractive web pages.

Static Files in Django

Static files include:

- CSS
- JavaScript
- Images

Django manages these files using the **static** directory.

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

- Static files are files that do not change dynamically. These include **CSS**, **JavaScript**, and **images**. Django provides a built-in system to manage and serve static files efficiently during development and production.

What are Static Files?

Static files are used for:

- Page design (CSS)
- Client-side logic (JavaScript)
- Images, icons, fonts

Static Files Setup in Django

1. Add Static Settings

In settings.py:

STATIC_URL = '/static/'
2. Create Static Folder Structure

```
project/
└── app/
    └── static/
        └── app/
            ├── css/
            │   └── style.css
            ├── js/
            │   └── script.js
            └── images/
```

3. Load Static Files in Template

At the top of HTML file:

{% load static %}
4. Use Static Files in HTML

CSS:

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

JavaScript:

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

Image:

<img src="{% static 'app/images/logo.png' %}">

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

- JavaScript is used to make web pages interactive on the client side. In Django, JavaScript is integrated with HTML templates to handle user actions like button clicks, form validation, and dynamic content updates without reloading the page.

Role of JavaScript in Django

Django handles backend logic, while JavaScript works on the browser side to:

- Improve user experience
- Handle events
- Update content dynamically
- Reduce server requests

Adding JavaScript to Django Templates

1. Create JavaScript File

```
app/
└── static/
    └── app/
        └── js/
            └── script.js
```

2. Write JavaScript Code

```
function showMessage() {
   alert("Button Clicked!");
}
```

3. Load Static Files in Template

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

4. Link JavaScript in HTML

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

6. inking external or internal JavaScript files in Django.

- JavaScript is used to add interactivity to web pages. In Django, JavaScript can be linked in two ways:

1. **External JavaScript files**
2. **Internal JavaScript (inside HTML templates)**

Django uses **static files** to manage external JavaScript files.

1. Linking External JavaScript Files in Django

Step 1: Create JavaScript File

```
app/
 └── static/
     └── app/
         └── js/
             └── script.js
```

Step 2: Write JavaScript Code

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

Step 3: Load Static Files in Template

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

Step 4: Link JavaScript File

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

Step 5: Use JavaScript Function

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

2. Using Internal JavaScript in Django Template

Internal JavaScript is written directly inside the HTML file.

Example

```
<script>
   function showMessage() {
      alert("Hello from internal JavaScript!");
   }
</script>

<button onclick="showMessage()">Click Here</button>
```

7. Overview of Django: Web development framework.

- Django is a high-level **Python web development framework** used to build secure, scalable, and dynamic websites quickly. It follows the principle of **"Don't Repeat Yourself (DRY)"** and helps developers create web applications with less code.

What is Django?

Django is an **open-source framework** written in Python. It provides built-in tools and libraries that handle common web development tasks such as database management, user authentication, and URL routing.

Features of Django

- Fast development
- Secure by default
- Built-in admin panel
- Object Relational Mapping (ORM)
- Template system
- URL routing system

Django Architecture (MVT)

Django follows the **MVT architecture**:

- **Model** – Handles database data
- **View** – Contains business logic
- **Template** – Handles HTML presentation

Applications of Django

- Content Management Systems
- E-commerce websites
- Social media platforms
- Management systems
- REST APIs

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

- Django is a powerful Python web framework used to build modern web applications. It provides many built-in features that make development faster, more secure, and scalable.

1. Scalability

Django is designed to handle large and complex applications.

- Supports high traffic websites
- Easy to scale by adding servers
- Used by big platforms like Instagram and Disqus

2. High Security

Django provides strong security features by default:

- Protection against SQL Injection
- Cross-Site Scripting (XSS) protection
- Cross-Site Request Forgery (CSRF) protection
- Secure authentication system

3. Fast Development

- Built-in tools reduce coding time
- Ready-to-use admin panel
- Reusable components

4. Clean and Structured Code

- Follows MVT architecture
- Separation of logic and design
- Easy to maintain and update

5. Built-in Admin Panel

- Automatic admin interface
- Easy data management
- No extra coding required

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

- Django and Flask are popular **Python web frameworks**. Both are used to build web applications, but they differ in size, features, and use cases. Choosing the right framework depends on project requirements and developer experience.

What is Django?

Django is a **full-stack web framework**. It provides many built-in features like authentication, admin panel, ORM, and security tools.

Best for: Large and complex applications.

What is Flask?

Flask is a **micro web framework**. It provides basic tools and allows developers to add features as needed using extensions.

Best for: Small projects and learning.

Django vs Flask Comparison

Feature	Django	Flask
Framework type	Full-stack	Micro framework
Learning curve	Moderate to high	Easy
Built-in features	Many	Minimal
Admin panel	Yes	No

Feature	Django	Flask
Database handling	Built-in ORM	Uses extensions
Scalability	High	Moderate
Flexibility	Less	More
Development speed	Very fast for big apps	Fast for small apps

Which to Choose and Why

Choose Django if:

- You are building a large application
- You need built-in security and admin panel
- You want fast development with less configuration
- You are working on enterprise-level projects

Choose Flask if:

- You are a beginner
- You are building a small or simple app
- You want full control and flexibility
- You are creating APIs or prototypes

10. Understanding the importance of a virtual environment in Python projects.

- A virtual environment in Python is an isolated workspace that allows developers to install and manage project-specific libraries without affecting the system-wide Python installation. It is very important for maintaining clean, stable, and conflict-free Python projects.

What is a Virtual Environment?

A virtual environment is a separate Python environment that:

- Has its own Python interpreter
- Has its own installed packages
- Is independent from other projects

Each project can use different versions of libraries without problems.

Why Virtual Environment is Important

1. Avoids Dependency Conflicts

Different projects may require different versions of the same package.
A virtual environment keeps them separate.

2. Keeps System Python Clean

Packages installed inside a virtual environment do not affect the system Python, preventing system errors.

3. Improves Project Stability

The project runs the same way on different machines when the same dependencies are used.

4. Easy Dependency Management

Using `requirements.txt`, all project libraries can be installed easily.

5. Best Practice for Django and Flask

Frameworks like **Django** and **Flask** strongly recommend using virtual environments for safe and organized development.

11. Using venv or virtualenv to create isolated environments.

- In Python projects, different applications may require different versions of libraries. To avoid conflicts, Python provides tools like **venv** and **virtualenv** to create isolated environments. These environments help manage dependencies separately for each project.

What is an Isolated Environment?

An isolated environment is a separate Python workspace where:

- Packages are installed only for that project
- Other projects are not affected
- System Python remains safe

Using venv

venv is a built-in module available in Python 3.

Create Virtual Environment
python -m venv venv
Activate Virtual Environment

Windows:

venv\Scripts\activate

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

- Django is a Python web framework that follows a project–app structure.
  A **project** is the main container, and **apps** are individual modules that perform specific tasks like authentication, events, or payments.

Prerequisites

- Python installed
- Virtual environment activated
- Django installed

pip install django

Steps to Create a Django Project

Step 1: Create Django Project
django-admin startproject myproject

Step 2: Move into Project Folder
cd myproject

Step 3: Run Development Server
python manage.py runserver

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

- Django follows a structured architecture to manage web applications efficiently. Three important files in a Django project are **manage.py**, **urls.py**, and **views.py**. Each file has a specific role in running and controlling the application.

1. Role of manage.py

What is manage.py?

manage.py is a command-line utility used to interact with a Django project.

Functions of manage.py

- Starts the development server
- Creates apps
- Applies database migrations
- Runs administrative commands

2. Role of urls.py

What is urls.py?

urls.py maps URLs to views. It decides **which view runs when a user visits a specific URL**.

Purpose

- URL routing
- Clean and readable URLs
- Connects browser requests to views

3. Role of views.py

What is views.py?

views.py contains the **business logic** of the application.
It processes user requests and returns responses.

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

- Django follows the **MVT (Model–View–Template)** architecture. This structure helps in separating data handling, business logic, and user interface, making web applications clean, organized, and easy to maintain.

What is MVT Architecture?

MVT stands for:

- **Model**
- **View**
- **Template**

Each component has a specific role in the Django framework.

1. Model

- Handles database structure and data
- Defines tables using Python classes
- Interacts with the database using ORM

Example:

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
```

2. View

- Contains application logic
- Receives user requests
- Fetches data from models
- Sends data to templates

Example:

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

def student_list(request):
    students = Student.objects.all()
    return render(request, 'students.html', {'students': students})
```

3. Template

- Handles the user interface
- Displays dynamic data using HTML
- Uses Django Template Language (DTL)

Example:

```
<ul>
{% for student in students %}
    <li>{{ student.name }}</li>
{% endfor %}
</ul>
```

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

- Django provides a powerful **built-in admin panel** that allows developers and administrators to manage application data easily. It is automatically generated and helps perform database operations without writing extra code.

What is Django Admin Panel?

The Django admin panel is a **web-based interface** used to:

- Add, update, delete, and view data
- Manage users and permissions
- Control database records

It is available by default when a Django project is created.

Features of Django Admin Panel

- Ready-to-use interface
- Secure login system
- Automatic form generation

- Supports all models
- Easy customization

Enabling Admin Panel

1. Create Superuser

```
python manage.py createsuperuser
```

2. Register Model in admin.py

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

admin.site.register(Student)
```

3. Access Admin Panel

Open browser and go to:

```
http://127.0.0.1:8000/admin/
```

16. Customizing the Django admin interface to manage database records.

- Django provides a built-in admin panel to manage database records. By default, it is simple, but Django allows developers to **customize the admin interface** to make data management easier, faster, and more user-friendly.

Why Customize the Admin Interface?

- Better data visibility
- Easy record management
- Faster searching and filtering
- Improved user experience for admins

Registering a Model

Before customization, a model must be registered.

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

admin.site.register(Student)
```

Using ModelAdmin for Customization

1. Display Fields (list_display)

Shows selected fields in the admin list view.

```
class StudentAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'email')
```

admin.site.register(Student, StudentAdmin)

2. Search Records (search_fields)

Allows searching records easily.

```
class StudentAdmin(admin.ModelAdmin):
    search_fields = ('name', 'email')
```

3. Filter Records (list_filter)

Adds filters on the right side.

```
class StudentAdmin(admin.ModelAdmin):
    list_filter = ('course',)
```

4. Pagination (list_per_page)

Controls records per page.

```
class StudentAdmin(admin.ModelAdmin):
    list_per_page = 10
```

5. Ordering Records (ordering)

Sorts records automatically.

```
class StudentAdmin(admin.ModelAdmin):
    ordering = ('name',)
```

6. Read-Only Fields

Prevents editing important fields.

```
class StudentAdmin(admin.ModelAdmin):
    readonly_fields = ('created_at',)
```

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

- In Django, **URL routing** is used to connect a web address (URL) to a specific function or class called a **view**. This work is done using the urls.py file. It helps Django decide which view should handle a user request.

What is urls.py?

urls.py is a configuration file that:

- Defines URL patterns
- Maps URLs to views
- Controls navigation in a Django application

Basic Structure of urls.py

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

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

Creating URL Patterns

1. Import Required Modules

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

2. Define URL Pattern

```
path('about/', views.about, name='about')
```

- 'about/' → URL path
- views.about → View function
- name='about' → URL name for templates

Linking URLs with Views

views.py

```
def home(request):
   return HttpResponse("Home Page")
```

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

- In Django, dynamic web pages are created by connecting **views** with **templates**.
  Views handle the logic and data, while templates display that data as HTML. This integration allows Django to render dynamic content based on user requests.

Role of Views and Templates

- **View**: Processes requests and prepares data
- **Template**: Displays data using HTML and template tags

Steps to Integrate Templates with Views

1. Create a Template

Create an HTML file inside the templates folder.

```
app/
└── templates/
    └── home.html
```

home.html

```
<!DOCTYPE html>
<html>
<body>
   <h1>Welcome {{ name }}</h1>
</body>
</html>
```

2. Create a View

Define a view in views.py that sends data to the template.

```
from django.shortcuts import render

def home(request):
   return render(request, 'home.html', {'name': 'User'})
```

3. Configure Template Settings

In settings.py:

```
TEMPLATES = [
   {
      'DIRS': [BASE_DIR / 'templates'],
   },
]
```

4. Map URL to View

In urls.py:

```
path('', views.home, name='home')
```

19. Using JavaScript for front-end form validation.

- Form validation ensures that the data entered by the user is correct before sending it to the server. Using **JavaScript** on the front-end helps check the data in real-time, improves user experience, and reduces unnecessary server requests.

Why Front-End Validation is Important

- Prevents invalid data submission
- Provides instant feedback to users

- Reduces server load
- Improves overall user experience

Steps for Front-End Validation

1. Create HTML Form

```
<form id="myForm" onsubmit="return validateForm()">
   Name: <input type="text" id="name"><br>
   Email: <input type="text" id="email"><br>
   <input type="submit" value="Submit">
</form>
<p id="error-msg" style="color:red;"></p>
```

2. Write JavaScript Validation Function

```
function validateForm() {
   let name = document.getElementById('name').value;
   let email = document.getElementById('email').value;
   let errorMsg = "";

   if(name == "") {
      errorMsg += "Name is required. ";
   }
   if(email == "" || !email.includes("@")) {
      errorMsg += "Enter a valid email.";
   }

   if(errorMsg != "") {
      document.getElementById('error-msg').innerText = errorMsg;
      return false; // Prevent form submission
   }
   return true; // Submit form if valid
}
```

3. How it Works

1. User clicks submit
2. validateForm() function checks each input
3. If invalid, shows error message and stops submission
4. If valid, allows the form to submit to the server

20. Connecting Django to a database (SQLite or MySQL).

- Django supports multiple databases to store and manage application data. By default, Django uses **SQLite**, but it can also work with databases like **MySQL**, **PostgreSQL**, and **Oracle**. Connecting Django to a database allows the application to store, retrieve, and manage data efficiently.

1. Using SQLite (Default Database)

SQLite comes pre-configured with Django.

Settings in settings.py

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

Steps

1. Create Django project
2. Run migrations to create database tables

python manage.py migrate

3. Database file db.sqlite3 is created automatically

2. Using MySQL Database

Step 1: Install MySQL and Connector

pip install mysqlclient

Step 2: Configure Database in settings.py

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'mydatabase',
      'USER': 'root',
      'PASSWORD': 'password123',
      'HOST': 'localhost',
      'PORT': '3306',
   }
}
```

Step 3: Run Migrations

python manage.py migrate

Step 4: Test Database Connection

- Create superuser

python manage.py createsuperuser

- Run server

python manage.py runserver

21. Using the Django ORM for database queries.

- Django provides a powerful Object-Relational Mapping (ORM) system that allows developers to interact with the database using Python code instead of writing raw SQL. The ORM simplifies database operations and makes the code cleaner and more maintainable.

What is Django ORM?

- Converts Python classes (models) into database tables
- Allows querying database using Python methods
- Supports CRUD operations (Create, Read, Update, Delete)
- Works with multiple databases (SQLite, MySQL, PostgreSQL)

Creating a Model

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    age = models.IntegerField()
```

- Each model represents a table
- Each attribute represents a column

Basic ORM Queries

1. Create Records

```
student = Student(name="John", email="john@example.com", age=20)
student.save()
```

2. Retrieve Records

```
all_students = Student.objects.all()
john = Student.objects.get(name="John")
young_students = Student.objects.filter(age__lt=25)
```

3. Update Records

```
john.email = "john123@example.com"
john.save()
```

4. Delete Records

```
john.delete()
```

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

- Django's **ORM (Object-Relational Mapping)** allows developers to interact with the database using Python code instead of SQL. ORM maps Python **models** to database tables, making data handling simple and secure. **QuerySets** are the way Django retrieves and manipulates database records.

1. What is a QuerySet?

A **QuerySet** is a collection of database objects retrieved from the database using Django ORM. It allows filtering, ordering, and modifying data.

Characteristics

- Lazy: QuerySet hits the database only when needed
- Chainable: Multiple filters can be applied
- Iterable: Can loop through results

2. Creating a Model

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField()
```

3. Using QuerySets

a) Retrieve All Records

```
students = Student.objects.all()
```

b) Filter Records

```
young_students = Student.objects.filter(age__lt=25)
```

c) Get a Single Record

```
john = Student.objects.get(name="John")
```

d) Exclude Records

```
older_students = Student.objects.exclude(age__lt=25)
```

23. Using Django's built-in form handling.

- Django provides a **built-in forms framework** to simplify creating and managing HTML forms. It helps handle user input, validate data, and save it to the database efficiently and securely.

Why Use Django Forms?

- Automatically generate HTML forms
- Validate user input
- Prevent invalid data submission
- Secure against common attacks like XSS and CSRF
- Integrates easily with models

1. Creating a Form Using forms.py

```
from django import forms
from .models import Student

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
```

```
        fields = ['name', 'email', 'age']
```

- forms.ModelForm links the form to a model
- fields specifies which model fields to include

2. Rendering the Form in a View

```
from django.shortcuts import render, redirect
from .forms import StudentForm

def add_student(request):
    if request.method == "POST":
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('home')
    else:
        form = StudentForm()
    return render(request, 'add_student.html', {'form': form})
```

3. Displaying Form in a Template

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>
```

- {{ form.as_p }} renders fields as paragraphs
- {% csrf_token %} ensures security

4. Validating Form Data

- form.is_valid() checks whether the submitted data is correct
- Automatic validation for:
    - Required fields
    - Field types (e.g., email, number)
    - Custom validation (optional)

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

- Django provides a **built-in authentication system** to manage users, login, logout, and passwords securely. This system handles common security tasks and allows developers to quickly implement user management.

1. Creating a User (Sign Up)

a) Using Django’s Built-in User Model

```
from django.contrib.auth.models import User

# Create a user
user = User.objects.create_user(username='john', email='john@example.com', password='password123')
user.save()
```

b) Using a Sign-Up Form

```
from django import forms
from django.contrib.auth.models import User

class SignUpForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ['username', 'email', 'password']
```

2. Logging In a User

View for Login

```
from django.contrib.auth import authenticate, login
from django.shortcuts import render, redirect

def login_view(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home')
        else:
            return render(request, 'login.html', {'error': 'Invalid credentials'})
    return render(request, 'login.html')
```

3. Logging Out a User

```
from django.contrib.auth import logout
from django.shortcuts import redirect

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

4. Password Management

a) Changing Password

```
from django.contrib.auth.forms import PasswordChangeForm
from django.contrib.auth import update_session_auth_hash

def change_password(request):
    if request.method == 'POST':
```

```
        form = PasswordChangeForm(request.user, request.POST)
        if form.is_valid():
            user = form.save()
            update_session_auth_hash(request, user)  # Keep user logged in
            return redirect('home')
    else:
        form = PasswordChangeForm(request.user)
    return render(request, 'change_password.html', {'form': form})
```

b) Password Reset

- Django provides **built-in views** for sending reset emails and updating passwords securely.

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

- AJAX (**Asynchronous JavaScript and XML**) allows web pages to **communicate with the server without reloading the page**. In Django, AJAX is used to send and receive data dynamically, improving user experience.

Why Use AJAX?

- Avoid full-page reloads
- Faster and smoother user experience
- Update only part of the page
- Useful for forms, search suggestions, and real-time updates

1. Setting Up AJAX with Django

a) Create a View

```
from django.http import JsonResponse

def check_username(request):
    username = request.GET.get('username')
    exists = User.objects.filter(username=username).exists()
    return JsonResponse({'exists': exists})
```

b) URL Mapping

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

urlpatterns = [
    path('check-username/', views.check_username,
name='check_username'),
]
```

c) HTML Form

```
<input type="text" id="username" placeholder="Enter Username">
<span id="msg"></span>
```

d) AJAX Request Using JavaScript

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script>
    $('#username').on('keyup', function() {
        let username = $(this).val();
        $.ajax({
            url: '/check-username/',
            data: { 'username': username },
            dataType: 'json',
            success: function(data) {
                if(data.exists) {
                    $('#msg').text('Username already taken');
                } else {
                    $('#msg').text('Username available');
                }
            }
        });
    });
</script>
```

26. Techniques for customizing the Django admin panel.

- Django provides a built-in admin panel for managing database records. By default, it is simple, but developers can **customize it** to improve usability, appearance, and data management efficiency.

Why Customize the Admin Panel?

- Make data easier to view
- Improve admin workflow
- Enable quick search and filtering
- Control user access and permissions

Techniques for Customization

1. Register Models with ModelAdmin

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

class StudentAdmin(admin.ModelAdmin):
   list_display = ('id', 'name', 'email')
   search_fields = ('name', 'email')
```

```
    list_filter = ('course',)
    ordering = ('name',)
    list_per_page = 10

admin.site.register(Student, StudentAdmin)
```

2. Customize Field Layouts

- Group fields using fieldsets

```
class StudentAdmin(admin.ModelAdmin):
    fieldsets = (
        ('Personal Info', {'fields': ('name', 'email')}),
        ('Academic Info', {'fields': ('course', 'age')}),
    )
```

3. Make Fields Read-Only

```
class StudentAdmin(admin.ModelAdmin):
    readonly_fields = ('created_at',)
```

4. Add Actions

- Perform bulk operations

```
def make_graduated(modeladmin, request, queryset):
    queryset.update(status='Graduated')

class StudentAdmin(admin.ModelAdmin):
    actions = [make_graduated]
```

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

- In modern web applications, accepting online payments is essential. Django allows integration with payment gateways like Paytm, Razorpay, Stripe, or PayPal. This enables users to pay securely online, and the application can process and store payment information efficiently.

What is a Payment Gateway?

A payment gateway is a service that:

- Processes online transactions
- Verifies payment details
- Transfers funds from the customer to the business account securely

Steps to Integrate Paytm in Django

1. Create a Paytm Merchant Account

- Sign up on the Paytm developer portal
- Get Merchant ID, Merchant Key, and Website Name

2. Install Paytm SDK or Library
pip install paytmchecksum

- paytmchecksum helps create secure transaction checksums

3. Create Payment Model

```
from django.db import models

class Payment(models.Model):
    order_id = models.CharField(max_length=100)
    amount = models.FloatField()
    status = models.CharField(max_length=50, default='Pending')
    created_at = models.DateTimeField(auto_now_add=True)
```

4. Create a Payment View

```
from django.shortcuts import render, redirect
from paytmchecksum import PaytmChecksum
from .models import Payment

def paytm_payment(request):
    payment = Payment.objects.create(order_id="ORDER123", amount=100)
    param_dict = {
        'MID': 'YourMerchantID',
        'ORDER_ID': payment.order_id,
        'CUST_ID': 'customer001',
        'TXN_AMOUNT': str(payment.amount),
        'CHANNEL_ID': 'WEB',
        'WEBSITE': 'WEBSTAGING',
        'INDUSTRY_TYPE_ID': 'Retail',
        'CALLBACK_URL': 'http://localhost:8000/callback/',
    }
    param_dict['CHECKSUMHASH'] = PaytmChecksum.generateSignature(param_dict,
'YourMerchantKey')
    return render(request, 'paytm_redirect.html', {'param_dict': param_dict})
```

5. Handle Payment Callback

- Paytm will send transaction status to CALLBACK_URL
- Update payment status in the database

```
def payment_callback(request):
    response = request.POST
    status = response['STATUS']
```

```
    order_id = response['ORDERID']
    payment = Payment.objects.get(order_id=order_id)
    payment.status = status
    payment.save()
    return render(request, 'payment_status.html', {'status': status})
```

6. Frontend Integration

```
<form method="post" action="{% url 'paytm_payment' %}">
    {% csrf_token %}
    <button type="submit">Pay Now</button>
</form>
```

28. Steps to push a Django project to GitHub.

- Version control is important in software development. **GitHub** is a platform for hosting Git repositories. Pushing a Django project to GitHub helps in **backup, collaboration, and version tracking**.

Prerequisites

- Git installed
- GitHub account
- Django project ready

Step 1: Initialize Git Repository

Open terminal in your project folder and run:

```
git init
```

Step 2: Create .gitignore File

Exclude unnecessary files (like venv, __pycache__, db.sqlite3) from Git.

Example .gitignore:

```
venv/
*.pyc
__pycache__/
db.sqlite3
*.log
```

Step 3: Add Files to Git

```
git add .
```

- . adds all project files except those in .gitignore

Step 4: Commit Changes

git commit -m "Initial commit"

Step 5: Create Repository on GitHub

- Go to GitHub → New Repository
- Give a **name**, description (optional), and create it
- Copy the **repository URL** (HTTPS or SSH)

Step 6: Add Remote Repository

git remote add origin https://github.com/username/repository.git

Step 7: Push Project to GitHub

git branch -M main
git push -u origin main

- -M main renames default branch to main (if needed)
- -u origin main sets upstream for future pushes

Step 8: Verify

- Go to your GitHub repository
- All project files should be visible

29. Introduction to deploying Django projects to live servers like PythonAnywhere.

- After developing a Django project locally, the next step is to **deploy it on a live server** so that users can access it via the internet. **PythonAnywhere** is a popular cloud platform that allows hosting Python and Django applications easily.

Why Deploy Django Projects?

- Make the website accessible online
- Test the project in a real environment
- Share with clients or users
- Use cloud resources without setting up your own server

Steps to Deploy Django Project on PythonAnywhere

1. Create an Account

- Sign up at PythonAnywhere
- Free or paid account based on your needs

2. Upload Your Django Project

- Use **Git** to clone your project:

git clone https://github.com/username/project.git

- Or upload files manually via the PythonAnywhere dashboard

3. Set Up a Virtual Environment
mkvirtualenv myenv --python=python3.11
pip install django
pip install -r requirements.txt

4. Configure the Project

- Update settings.py:
  - Set ALLOWED_HOSTS = ['yourusername.pythonanywhere.com']
  - Set DEBUG = False

5. Set Up Static Files

- Collect static files:

python manage.py collectstatic

- Configure static file paths in PythonAnywhere web app settings

6. Configure WSGI File

- Edit the WSGI configuration file in PythonAnywhere
- Point it to your Django project and virtual environment

7. Run Database Migrations
python manage.py migrate

8. Access Your Live Website

- Go to https://yourusername.pythonanywhere.com
- Your Django project should now be live

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

- Social login allows users to sign up or log in to a Django website using accounts from **Google, Facebook, GitHub**, etc. This simplifies registration, reduces password management, and provides a secure authentication process using **OAuth2**.

Why Use Social Login?

- Users don’t need to remember new passwords
- Faster registration and login
- Secure authentication managed by social platforms
- Enhances user experience

Steps to Set Up Social Login in Django

1. Install Required Packages

Use **django-allauth**, which simplifies social authentication:

pip install django-allauth

2. Add Installed Apps

In settings.py:

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

3. Configure Site ID

```
SITE_ID = 1
```

4. Update Authentication Backends

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

5. Configure URLs

```
from django.urls import path, include

urlpatterns = [
    path('accounts/', include('allauth.urls')),  # Social login URLs
]
```

6. Set Up OAuth Credentials

- **Google**: Create project in Google Developers Console → get Client ID & Secret
- **Facebook**: Create app in Facebook Developers → get App ID & Secret
- **GitHub**: Create OAuth app in GitHub Settings → get Client ID & Secret

Add credentials in Django admin under **Social Applications**.

7. Template Integration

```
<a href="{% provider_login_url 'google' %}">Login with Google</a>
<a href="{% provider_login_url 'facebook' %}">Login with Facebook</a>
<a href="{% provider_login_url 'github' %}">Login with GitHub</a>
```

31. Integrating Google Maps API into Django projects.

- Google Maps API allows developers to embed interactive maps into web applications. Integrating Google Maps in Django helps display **locations, markers, routes**, and other map features directly on your website.

Why Integrate Google Maps?

- Show business locations or landmarks
- Display real-time routes and directions
- Enhance user interaction and UI
- Useful for location-based services

Steps to Integrate Google Maps in Django

1. Get Google Maps API Key

- Go to Google Cloud Console
- Create a new project → Enable **Maps JavaScript API**
- Generate **API key**

2. Create a Template

Example map.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Google Map</title>
    <script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY"></script>
```

```
    <script>
      function initMap() {
        var location = {lat: 23.0225, lng: 72.5714}; // Example coordinates
        var map = new google.maps.Map(document.getElementById('map'), {
          zoom: 12,
          center: location
        });
        var marker = new google.maps.Marker({position: location, map: map});
      }
    </script>
  </head>
  <body onload="initMap()">
    <h3>Our Location</h3>
    <div id="map" style="height:400px;width:100%;"></div>
  </body>
</html>
```

3. Create a View

In views.py:

```
from django.shortcuts import render

def map_view(request):
    return render(request, 'map.html')
```

4. Map URL to View

In urls.py:

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

urlpatterns = [
    path('map/', views.map_view, name='map'),
]
```

5. Optional: Dynamic Locations

- Pass coordinates from Django view to template:

```
def map_view(request):
    context = {'lat': 23.0225, 'lng': 72.5714}
    return render(request, 'map.html', context)
```

Template:

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