

so tell me steps i will delete and now start
ChatGPT said:

Perfect! Let's start fresh and create your Django projects the right way – each with its own virtual environment and folder. Follow these steps:

✂ Step 0: Delete old folders

; If you want to delete everything and start clean:

```
cd ~/Desktop/INT253\ DJANGO/  
rm -rf testenv first
```

(Only do this if you don't need your first project anymore)

; 📁 Step 1: Create a folder for your project

```
cd ~/Desktop/INT253\ DJANGO/  
mkdir FirstProject  
cd FirstProject
```

; 🐍 Step 2: Create virtual environment

```
python3 -m venv venv
```

; ⚡ Step 3: Activate the virtual environment

```
source venv/bin/activate
```

; You'll see your prompt change to:

```
(venv) sanuj ~/Desktop/INT253 DJANGO/FirstProject $
```

; 📦 Step 4: Install Django inside the venv

```
pip install django
```

; 🚀 Step 5: Start your Django project

```
django-admin startproject first .
```

The dot . creates the project in the current folder instead of adding an extra subfolder.

; 6. Run Migrations

```
python manage.py migrate
```

; 🖋 Step 7: Run the server

```
python manage.py runserver
```

```
; Your Django app will now run at: // till here for creating first project
http://127.0.0.1:8000
```

```
; Create Your First App (If Not Done Yet)
```

```
; A Django project can have multiple apps. Run this to create app:
```

```
python manage.py startapp myapp //Recommended cmd to create app
```

```
django-admin startapp myapp // Alternate cmd to create app
```

Now you'll see a new folder myapp/ inside firstProject.

```
; 🔄 To Exit a Virtual Environment:
```

```
deactivate
```

✅ Recommended structure (per-project isolation):

Create a folder for each project, like this:

```
~/Desktop/INT253 DJANGO/
```

```
|
|— FirstProject/
|   |— venv/           ← your virtual environment (optional but recommended)
|   |— first/         ← Django *project folder* (configuration)
|   |   |— myapp/      ← Django *app folder* (your logic)
|   |   |— manage.py/
|
|— SecondProject/
|   |— venv/           ← another separate env
|   |— second/        ← second project code
```

```
; ✅ Final Summary
```

Folder	Purpose
--------	---------

first/	Django project (settings, routing, config)
--------	--

myapp/	Django app (business logic, views, models, etc.)
--------	--

firstProject/	Your root project folder (can include env, README, etc.)
---------------	--

Got it 👍 Since your exam is tomorrow and this is a viva for Django, I'll explain your syllabus **step by step** in simple language so you can remember key points and answer confidently.

◆ Django Viva Preparation (INT253)

Unit I – Introduction to Django

- **Django** → A high-level Python web framework for rapid development, secure and scalable apps.
- **Install:**

```
bash
```

[Copy code](#)

```
pip install django
```

- **Start project:**

```
bash
```

[Copy code](#)

```
django-admin startproject myproject  
cd myproject  
python manage.py runserver
```

- **Start app:**

```
bash
```

[Copy code](#)

```
python manage.py startapp myapp
```


- **Project vs App:**
 - **Project** = The whole website (settings, multiple apps, configs).
 - **App** = A module providing specific functionality (e.g., blog, shop).
- **Files:**
 - `manage.py` – Run commands.
 - `settings.py` – Configurations.
 - `urls.py` – URL mappings.
 - `wsgi.py` / `asgi.py` – Deployment entry points.

Unit II – Views & URLs

Unit II – Views & URLs

- **View** = Python function/class that returns a response (HTML, JSON, etc.).


python

 Copy code

```
from django.http import HttpResponse
def home(request):
    return HttpResponse("Hello Django")
```

- **URLs** → Connects path to view.

python

 Copy code

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


urlpatterns = [
    path("", views.home, name="home"),
    path("user/<int:id>/", views.user_profile),
]
```

- **Requests:** GET, POST, PUT, DELETE.
- **Error Handling:**
 - **404** – Page not found.
 - **500** – Server error.

Unit III – Templates, Debugging & Testing

- **Template** = HTML with dynamic content using Django Template Language (DTL).
- **Example** (`index.html`):

html

 Copy code

```
<h1>Hello, {{ user_name }}</h1>
{% if is_logged_in %}
    <p>Welcome back!</p>
{% else %}
    <p>Please log in.</p>
{% endif %}
{% for item in items %}
    <li>{{ item }}</li>
{% endfor %}
```

- **Inheritance:**

html

 Copy code


```
<!-- base.html -->
<html><body>{% block content %}{% endblock %}</body></html>

<!-- child.html -->
{% extends "base.html" %}
{% block content %} <h1>Child Page</h1> {% endblock %}
```

- **Debugging:** Use `DEBUG = True` in `settings.py`. Django shows error details.

- **Debugging:** Use `DEBUG = True` in `settings.py`. Django shows error details.
- **Testing:**

bash


 Copy code

```
python manage.py test
```

Unit IV – Forms in Django

- **Forms** → Collect user input safely.
- **GET vs POST:**
 - GET → Data in URL.
 - POST → Data in request body (secure).
- **Example:**


python

 Copy code

```
from django import forms
class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
```

- **Views:**

python

 Copy code

```
def contact(request):
    if request.method == "POST":
        form = ContactForm(request.POST)
        if form.is_valid():
            return HttpResponseRedirect("Form submitted")
    else:
        form = ContactForm()
    return render(request, "contact.html", {"form": form})
```

- **CSRF:** Security token to protect from fake requests. `{% csrf_token %}` inside form.
- **Validation:** Django automatically checks field rules (email format, required, etc.).

Unit V – Models, Migrations & Admin

- **Model** = Python class that defines a database table.

Unit V – Models, Migrations & Admin

- **Model** = Python class that defines a database table.

```
python

from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

[Copy code](#)

- **Migrations:**

```
bash

python manage.py makemigrations
python manage.py migrate
```

[Copy code](#)

- **ORM (Object Relational Mapper):**

```
python

Student.objects.create(name="Alex", age=21)
Student.objects.filter(age__gt=18)
```

[Copy code](#)

- **Admin:**

- Create superuser:

```
bash

python manage.py createsuperuser
```

[Copy code](#)

- Register models in `admin.py`.

```
python

from .models import Student
admin.site.register(Student)
```

[Copy code](#)

Unit VI – Cookies, Sessions & Authentication

- **Cookies:** Small data stored in browser.

```
python

response = HttpResponse("Set Cookie")
response.set_cookie("name", "DjangoUser")
```

[Copy code](#)

- **Sessions:** Server-side data (linked to cookies).

```
python

request.session["user_id"] = 101
```

[Copy code](#)

- **Authentication:**

- Built-in user model: `from django.contrib.auth.models import User`
- Login/Logout:

- **Authentication:**

- Built-in user model: `from django.contrib.auth.models import User`
- Login/Logout:

```
python
```

[Copy code](#)

```
from django.contrib.auth import authenticate, login, logout
```

- **Protect views:**

```
python
```

[Copy code](#)

```
from django.contrib.auth.decorators import login_required
@login_required
def dashboard(request):
    return HttpResponse("Welcome")
```

◆ Key Course Outcomes (COs)

- **CO1:** Setup Django projects, apps, commands.
- **CO2:** Views, URLs, requests, error handling.
- **CO3:** Templates, loops, conditions, inheritance, testing.
- **CO4:** Forms, POST/GET, CSRF, validation.
- **CO5:** Models, migrations, ORM, Admin, database.
- **CO6:** Cookies, sessions, authentication.

✔ Quick Tips for Viva

- If asked "What is Django?" → "It's a Python web framework for rapid and secure development."
- If asked "Project vs App?" → "Project is the whole site, App is a specific module inside it."
- If asked "Why CSRF?" → "To protect against cross-site request forgery."
- If asked "What is ORM?" → "It lets us interact with the database using Python classes instead of SQL."
- If asked "Difference between GET and POST?" → "GET sends data in URL, POST sends data in request body securely."

◆ 50 Django Viva Questions & Answers

Unit I – Basics of Django

1. What is Django?

A Python web framework for rapid, secure, and scalable web development.

2. Why use Django?

It's fast, secure, has ORM, built-in admin, and handles authentication, forms, sessions.

3. How do you install Django?

```
pip install django
```

4. Command to start a project?

```
django-admin startproject projectname
```

5. Command to start an app?

```
python manage.py startapp appname
```

6. What is the difference between project and app?

Project = whole site (settings, multiple apps).

App = module inside project (blog, shop, etc.).

7. What is manage.py?

A command-line utility for running tasks like migrations, running server, creating superuser.

8. Where do you define project-level settings?

In `settings.py`.

Unit II – Views & URLs

9. What is a view in Django?

A function or class that handles a request and returns a response.

10. How do you map URLs to views?

Using `urlpatterns` in `urls.py`.

11. What is the default HTTP request method?

GET.

12. Difference between GET and POST?

GET sends data in URL, POST sends data in request body (secure).

13. How do you pass parameters in URLs?

Example: `path('user/<int:id>/', views.user_profile)`

14. How do you handle 404 errors?

Django automatically shows `404.html` if a page is not found.

15. Which file stores URL mappings?

`urls.py`.

Unit III – Templates & Debugging



Unit III – Templates & Debugging

16. What is a template in Django?

HTML file with Django Template Language (DTL) for dynamic content.

17. How do you render a template in views?

```
return render(request, "index.html", context)
```

18. What are template tags?

Special syntax like `{% for %}`, `{% if %}`, `{% block %}`.

19. How do you use template inheritance?

Use `{% extends "base.html" %}` and `{% block content %}`.

20. What does `{{ variable }}` do in templates?

Prints the value of a context variable.

21. What is the purpose of `DEBUG = True`?

To show detailed error pages during development.

22. How do you test a Django app?

```
python manage.py test
```

Unit IV – Forms

23. What are Django forms?

A way to handle user input with validation and security.

24. How do you create a form in Django?

By creating a class in `forms.py` that inherits from `forms.Form` or `forms.ModelForm`.

25. What is CSRF?

Cross-Site Request Forgery, an attack where fake requests are sent.

26. How do you prevent CSRF in Django?

By using `{% csrf_token %}` in forms.

27. What is the difference between `forms.Form` and `forms.ModelForm`?

- `Form` = manually define fields.
- `ModelForm` = automatically creates form fields from a model.

28. What happens if form data is invalid?

Django returns errors and does not save.

29. How do you redirect after form submission?

Using `HttpResponseRedirect` or `redirect()`.

Unit V – Models, Migrations & Admin

30. What is a Django model?

A Python class that defines a database table.

31. What is migration in Django?

Process of applying model changes to the database.

32. Command to create migrations?

```
python manage.py makemigrations
```

33. Command to apply migrations?

```
python manage.py migrate
```

34. How do you insert data into a model?

```
Student.objects.create(name="Alex", age=↓,
```

Unit V – Models, Migrations & Admin

30. What is a Django model?

A Python class that defines a database table.

31. What is migration in Django?

Process of applying model changes to the database.

32. Command to create migrations?

```
python manage.py makemigrations
```

33. Command to apply migrations?


```
python manage.py migrate
```

34. How do you insert data into a model?

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

35. How do you update a model record?

bash

 Copy code

```
student = Student.objects.get(id=1)
student.age = 22
student.save()
```

36. What is ORM?

Object Relational Mapper – lets us use Python instead of SQL.


37. What is the command to create a superuser?

```
python manage.py createsuperuser
```

38. How do you register a model in Django admin?

In `admin.py`:

python

 Copy code

```
from .models import Student
admin.site.register(Student)
```

39. What is the default database in Django?

SQLite.


Unit VI – Cookies, Sessions & Authentication

40. What is a cookie?

Small piece of data stored in browser.

41. How do you set a cookie in Django?

makefile

 Copy code

```
response = HttpResponse("Hello")
response.set_cookie("name", "User1")
```

42. What is a session in Django?

Server-side storage of user data linked to cookies.

43. How do you create a session?

```
request.session['user'] = "Alex"
```

44. How do you clear a session?

```
request.session.flush()
```




Unit VI – Cookies, Sessions & Authentication

40. What is a cookie?

Small piece of data stored in browser.

41. How do you set a cookie in Django?

makefile

 Copy code

```
response = HttpResponse("Hello")
response.set_cookie("name", "User1")
```

42. What is a session in Django?

Server-side storage of user data linked to cookies.

43. How do you create a session?

```
request.session['user'] = "Alex"
```

44. How do you clear a session?

```
request.session.flush()
```

45. What is the difference between cookies and sessions?

Cookies = stored in browser.


Sessions = stored on server.

46. How do you create a user in Django?

```
User.objects.create_user(username, email, password)
```

47. How do you authenticate a user?

sql

 Copy code

```
user = authenticate(username="alex", password="123")
if user:
    login(request, user)
```

48. How do you log out a user?

```
logout(request)
```

49. What decorator restricts a view to logged-in users?

```
@login_required
```

50. What is the default authentication system in Django?

Django provides a built-in `User` model and authentication framework.

✅ If you revise these 50 Q&As, you'll be able to confidently handle most viva questions.

Good question 🤔 In Django viva, examiners often ask about the folder layout and purpose of each file. Let's break down your given structure:

📁 Django Project Folder Layout Explained

pgsql

Copy code

```
FirstProject/
├── venv/           ← Virtual Environment (optional but recommended)
├── first/          ← Project folder (settings & configurations)
│   ├── __init__.py ← Marks directory as a Python package
│   ├── asgi.py     ← Entry point for ASGI servers (async deployment)
│   ├── settings.py ← Project settings (database, apps, middleware, etc.)
│   ├── urls.py     ← Project-level URL mappings (routes)
│   └── wsgi.py     ← Entry point for WSGI servers (deployment)
├── myapp/          ← Application folder (your actual app logic)
│   ├── __init__.py ← Marks directory as Python package
│   ├── admin.py    ← Register models for Django Admin
│   ├── apps.py     ← App configuration file
│   ├── migrations/ ← Database migration files (auto-created when models change)
│   │   └── __init__.py
│   ├── models.py   ← Database models (tables as Python classes)
│   ├── tests.py    ← Unit tests for your app
│   └── views.py     ← Business logic (functions/classes returning responses)
└── manage.py       ← Command-line utility to runserver, migrations, etc.
```

◆ Explanation of Each Part

1. venv/

- A virtual environment that isolates project dependencies (so different projects don't conflict).
- Activate it:

bash

Copy code

```
source venv/bin/activate # mac/linux
venv\Scripts\activate   # windows
```

2. first/ (Project Folder)

- Main configuration folder for your Django project.
- Important files:
 - `settings.py` → Contains database setup, installed apps, middleware, templates, static files, etc.
 - `urls.py` → Main URL dispatcher; routes requests to app `urls.py`.
 - `wsgi.py` → Deployment file for traditional web servers.

2. first/ (Project Folder)

- **Main configuration folder** for your Django project.
- Important files:
 - `settings.py` → Contains database setup, installed apps, middleware, templates, static files, etc.
 - `urls.py` → Main URL dispatcher; routes requests to app `urls.py`.
 - `wsgi.py` → Deployment file for traditional web servers.
 - `asgi.py` → Deployment file for async servers (WebSockets, async views).


3. myapp/ (Application Folder)

- Your **app code lives here** (models, views, forms, etc.).
- Files:
 - `models.py` → Defines database structure.
 - `views.py` → Contains view functions/classes for handling requests.
 - `admin.py` → Registers models in Django admin.
 - `apps.py` → App configuration class.
 - `migrations/` → Auto-generated files to sync DB with models.
 - `tests.py` → Write tests for app logic.

4. manage.py

- A shortcut script to interact with your project.
- Examples:

bash

 Copy code

```
python manage.py runserver      # Start server
python manage.py startapp blog  # Create new app
python manage.py makemigrations # Create migrations
python manage.py migrate       # Apply migrations
python manage.py createsuperuser # Create admin user
```

✅ How to remember for viva:

- **Project folder** (`first/`) = configurations.
- **App folder** (`myapp/`) = actual logic (models, views, templates, etc.).
- `manage.py` = command-line helper.
- `venv/` = environment for dependencies.

◆ WSGI vs ASGI in Django

1. WSGI (Web Server Gateway Interface)

- **File:** `wsgi.py`
- **Purpose:** Standard interface between **Python web applications and web servers**.
- **Use case:** Traditional synchronous web requests (HTTP).
- **Works with:** Servers like **Gunicorn, uWSGI, Apache mod_wsgi**.
- **Django default:** Before Django 3.0, only WSGI was used.

👉 In simple words:

- WSGI handles **normal request-response cycle** (page load, form submission).
- Suitable for **synchronous apps**.

2. ASGI (Asynchronous Server Gateway Interface)

- **File:** `asgi.py`
- **Purpose:** Newer standard supporting **asynchronous communication**.
- **Use case:** Real-time apps (chat, notifications, WebSockets, long polling).
- **Works with:** Servers like **Daphne, Uvicorn, Hypercorn**.
- **Django:** From version **3.0 onwards**, Django supports ASGI.

👉 In simple words:

- ASGI handles **async requests** (e.g., WebSockets for chat apps, live updates).
- It's **future-ready**, supports both sync + async.

👉 In simple words:

- ASGI handles **async requests** (e.g., WebSockets for chat apps, live updates).
- It's **future-ready**, supports both sync + async.

Location in Project

- Both files live in the **project folder** (`first/` in your case).
- Example (`asgi.py`):

python

📄 Copy code

```
import os
from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'first.settings')
application = get_asgi_application()
```

- Example (`wsgi.py`):

python

📄 Copy code

```
import os
from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'first.settings')
application = get_wsgi_application()
```

✅ Quick Comparison for Viva

Feature	WSGI	ASGI
Full form	Web Server Gateway Interface	Asynchronous Server Gateway Interface
Handles	Synchronous requests	Sync + Async (HTTP, WebSockets)
Django versions	Default before Django 3.0	Introduced from Django 3.0+
Use case	Normal websites	Real-time apps (chat, live notifications)
Server examples	Gunicorn, uWSGI, mod_wsgi	Daphne, Uvicorn, Hypercorn

⚡ One-liner for viva:

- *“WSGI is used for traditional synchronous requests, while ASGI supports both sync and async, making it suitable for modern real-time applications like chat apps and live notifications.”*

Perfect 👍 This is one of the most commonly asked viva questions: *"Explain the Django request-response lifecycle."*

I'll give you a **step-by-step flow** so you can answer confidently.

◆ Django Request-Response Lifecycle

👉 Step 1: User Request

- A user enters a URL in the browser (e.g., `http://127.0.0.1:8000/home/`).
- The request reaches **WSGI (or ASGI)** server.
 - WSGI → sync requests.
 - ASGI → async + sync requests.

👉 Step 2: URL Dispatcher (`urls.py`)

- Django looks at the `urls.py` file to find a matching URL pattern.
- If no match → returns **404 error**.

Example:

python

📄 Copy code

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

👉 Step 3: View (`views.py`)

- The matched URL calls the corresponding **view function/class**.
- Views handle business logic.
- Views may:
 - Return an **HttpResponse** directly.
 - Fetch data from the **Model**.
 - Pass data to **Template**.

Example:

python

📄 Copy code

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

👉 Step 4: Model (`models.py`) [if needed]

- If view needs data, it queries the **Model** (ORM → database).
- Models are Python classes mapped to database tables.

👉 Step 4: Model (`models.py`) [if needed]

- If view needs data, it queries the Model (ORM → database).
- Models are Python classes mapped to database tables.

Example:

python

📄 Copy code

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

👉 Step 5: Template Rendering (`templates/`)

- View passes data (context) to template.
- Django Template Language (DTL) dynamically generates HTML.

Example (`home.html`):

html

📄 Copy code

```
<h1>Hello, {{ name }}</h1>
```

👉 Step 6: Response Back to Browser

- Template is rendered into HTML.
- Django returns an `HttpResponse`.
- Browser displays the final web page to the user.

✅ Lifecycle Summary (One-Liner for Viva)

- *"Request comes → WSGI/ASGI → URL dispatcher (`urls.py`) → View (`views.py`) → Model (`models.py` , DB) → Template (HTML) → Response back to browser."*

⚡ Visual Flow:

cpp

📄 Copy code

```
wser → WSGI/ASGI → urls.py → views.py → models.py (DB) → template.html → Response → Browser
```

◆ What is `render` in Django?

- `render()` is a shortcut function provided by Django.
- It combines a template with a context dictionary and returns an `HttpResponse`.
- Saves you from writing `loader.get_template()` + `HttpResponse()` separately.

✓ Syntax:

python

Copy code

```
render(request, template_name, context=None, content_type=None, status=None, using=None)
```

- `request` → The HTTP request object.
- `template_name` → Path of the template (e.g., `"home.html"`).
- `context` → A dictionary of data to send to the template.
- `status` → (Optional) HTTP status code (e.g., 404, 500).

✓ Example:

views.py

python

Copy code

```
from django.shortcuts import render

def home(request):
    context = {"name": "Alex", "age": 21}
    return render(request, "home.html", context)
```

home.html

html

Copy code

```
<h1>Hello, {{ name }}! You are {{ age }} years old.</h1>
```

👉 Browser output:

sql

Copy code

```
Hello, Alex! You are 21 years old.
```

✓ One-liner for Viva:

"`render()` is a Django shortcut that loads a template, fills it with context data, and returns an `HttpResponse`."

◆ Cookies vs Sessions in Django

Feature	Cookies	Sessions
Storage	Stored on the client's browser.	Stored on the server (only session ID in client's cookie).
Security	Less secure (can be modified by user).	More secure (data is kept on server).
Data Size	Limited to ~4KB.	No strict limit (depends on server storage).
Lifespan	Can persist even after browser is closed (if not expired).	Lasts until user logs out, session expires, or browser is closed (default).
Use Case	Remembering small user preferences (theme, language, last login).	Storing sensitive user data (user ID, cart items, authentication).
Django Support	<code>response.set_cookie()</code> and <code>request.COOKIES</code> .	<code>request.session</code> .

✓ Example in Django

◆ Using Cookies

python

Copy code

```
def set_cookie(request):
    response = HttpResponseRedirect("Cookie Set")
    response.set_cookie("name", "Alex", max_age=3600) # 1 hour
    return response

def get_cookie(request):
    name = request.COOKIES.get("name")
    return HttpResponseRedirect(f"Hello {name}")
```

◆ Using Sessions

python

Copy code

```
def set_session(request):
    request.session["name"] = "Alex"
    return HttpResponseRedirect("Session Set")

def get_session(request):
    name = request.session.get("name", "Guest")
    return HttpResponseRedirect(f"Hello {name}")
```

✓ One-liner for Viva:

- "Cookies store data on the client-side, while sessions store data on the server-side. Sessions are more secure and used for authentication, while cookies are used for small preferences."

◆ What Django Does Internally with Sessions

- When you use `request.session["username"] = "Alex"`, Django:
 1. Creates a record in the database (`django_session` table).
 2. Generates a random session ID.
 3. Stores that ID in the browser as a cookie called `sessionid`.

So when the user visits again:

- Browser sends `sessionid` cookie → Django looks it up in DB → retrieves user data.

✓ Quick Viva Answer with Example:

"Cookies store user data in the browser (e.g., `response.set_cookie("name", "Alex")`), while sessions store data on the server and only keep a session ID in the cookie (e.g., `request.session["name"] = "Alex"`) making sessions more secure."

Great question 👍 Examiners often ask for **real-world examples** to see if you actually understand.
Let's compare **Cookies vs Sessions** with practical, real-world cases.

🍪 Cookies – Real World Example

- **"Remember Me" Checkbox on Login**
 - When you tick *"Remember Me"* while logging into Gmail/Facebook, a cookie is stored in your browser.
 - It remembers your login email/username or preference (like dark mode, language).
 - Even after you close and reopen the browser, the cookie is still there (until it expires).

👉 Django Example:

python

📄 Copy code

```
response.set_cookie("theme", "dark", max_age=86400) # 1 day
```

🔑 Sessions – Real World Example

- **Shopping Cart in Amazon/Flipkart**
 - When you add items to your cart, the cart items are stored in a **session**.
 - Even if you refresh the page, your cart is preserved (until you log out or session expires).
 - Actual cart data lives on the server, not in your browser.

👉 Django Example:

python

📄 Copy code

```
request.session["cart"] = ["Laptop", "Headphones", "Mouse"]
```

✅ Simple Analogy

- **Cookie = Hotel Room Key Card** 🗑️
 - You hold the card (browser), it unlocks your room. If lost, anyone can use it.
- **Session = Hotel Room** 🏠
 - Your actual belongings (data) are stored securely inside the room (server).
 - The card (cookie with sessionid) just tells the hotel which room belongs to you.

💠 One-Liner for Viva

- *"Cookies are used for storing small preferences on the client-side (e.g., Remember Me, themes), while sessions are used for storing secure, temporary user data on the server (e.g., shopping cart, login authentication)."*



◆ Django Imports Explained

1. `from django.http import HttpResponse`

- **What it is:**
 - `HttpResponse` is a class that Django uses to **send content back to the browser**.
- **Example:**

```
python Copy code  
  
from django.http import HttpResponse  
  
def home(request):  
    return HttpResponse("Hello, Django!")
```

- **Viva line:** *"HttpResponse is used to return HTML or plain text as a response from a view."*

2. `from django.urls import path`

- **What it is:**
 - `path()` is a function used to **map a URL pattern to a view**.
- **Example:**

```
python Copy code  
  
from django.urls import path  
from . import views  
  
urlpatterns = [  
    path('', views.home, name='home'),  
    path('about/', views.about, name='about'),  
]
```

- **Viva line:** *"path() connects a URL string to a view function so that Django knows what code to run for each URL."*

3. `from . import views`

- **What it is:**
 - Imports your `views.py` file from the current app folder.
 - Lets you call functions like `views.home()` or `views.about()`.
- **Viva line:** *"from . import views allows URL patterns to call functions defined in views.py."*

4. `from django import forms`

- **What it is:**
 - Imports Django's **form handling library**.
 - Lets you create **user input forms** with validation easily.
- **Example:**

```
python Copy code
```

4. `from django import forms`

- **What it is:**
 - Imports Django's **form handling library**.
 - Lets you create **user input forms** with validation easily.
- **Example:**

```
python Copy code  
  
from django import forms  
  
class ContactForm(forms.Form):  
    name = forms.CharField(max_length=100)  
    email = forms.EmailField()
```

- **Viva line:** “`forms` module is used to create and validate HTML forms in Django.”

5. URL Patterns (`urlpatterns`)

```
python Copy code  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', home, name='home'),  
    path('about/', about, name='about'),  
]
```

- **Explanation:**
 1. `path('admin/', admin.site.urls)` → URL for Django admin panel.
 2. `path('', home, name='home')` → Root URL (`http://127.0.0.1:8000/`) calls `home` view.
 3. `path('about/', about, name='about')` → URL `about/` calls `about` view.
- **Viva line:** “`urlpatterns` is a list of URL patterns that maps URLs to their respective view functions.”


✅ Summary Table for Viva

Line	Purpose	Copy
<code>from django.http import HttpResponse</code>	Return HTML or text as response from view	
<code>from django.urls import path</code>	Map URL patterns to views	
<code>from . import views</code>	Import views.py from current app	
<code>from django import forms</code>	Create & validate user forms	
<code>urlpatterns = [...]</code>	List of URLs mapped to views	

◆ 1. Django Variables


- **Syntax:** `{{ variable_name }}`
- **Purpose:** Display dynamic data passed from `views.py`.
- **Example:**

python

 Copy code

```
# views.py
def home(request):
    context = {"name": "Alex", "age": 21}
    return render(request, "home.html", context)
```


html

 Copy code

```
<!-- home.html -->
<h1>Hello, {{ name }}!</h1>
<p>You are {{ age }} years old.</p>
```

Output:

sql

 Copy code

```
Hello, Alex!
You are 21 years old.
```


◆ 2. Template Tags

- Definition: Special instructions inside `{% %}` for logic, loops, conditions, inheritance, etc.

Common Template Tags

Tag	Purpose	Example	📄
<code>{% if %}</code>	Conditional statements	<code>{% if age >= 18 %} Adult {% else %} Minor {% endif %}</code>	
<code>{% for %}</code>	Loop over lists	<code>{% for student in students %} {{ student.name }} {% endfor %}</code>	
<code>{% block %}</code>	Define blocks in base template	<code>{% block content %} ... {% endblock %}</code>	
<code>{% extends %}</code>	Inherit from base template	<code>{% extends "base.html" %}</code>	
<code>{% include %}</code>	Include another template	<code>{% include "header.html" %}</code>	
<code>{% url 'name' %}</code>	Reverse URL mapping	<code>Home</code>	
<code>{% csrf_token %}</code>	CSRF protection in forms	<code><form method="post">{% csrf_token %} </form></code>	
<code>{% empty %}</code>	Executes if loop is empty	<code>{% for i in items %}{{ i }}{% empty %}No Items{% endfor %}</code>	
<code>{% comment %}</code>	Add comments	<code>{% comment %} This is a comment {% endcomment %}</code>	
<code>{% static %}</code>	Load static files	<code></code>	

◆ 3. If-Else Statements


html

 Copy code

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}</p>
{% else %}
    <p>Please login</p>
{% endif %}
```

◆ 4. For Loops

html


 Copy code

```
<ul>
{% for student in students %}
    <li>{{ student.name }} - {{ student.age }}</li>
{% empty %}
    <li>No students available</li>
{% endfor %}
</ul>
```

◆ 5. Dynamic Templates


- Templates can display dynamic content from views or models.
- Example:

python

 Copy code

```
# views.py
def student_list(request):
    students = [{"name": "Alex"}, {"name": "John"}, {"name": "Mary"}]
    return render(request, "students.html", {"students": students})
```

html

 Copy code

```
<!-- students.html -->
<h2>Students</h2>
<ul>
{% for student in students %}
    <li>{{ student.name }}</li>
{% endfor %}
</ul>
```

◆ 6. Template Inheritance

- **Purpose:** Avoid repetition by creating a **base template**.

base.html

```
html

<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My Website{% endblock %}</title>
</head>
<body>
  {% include "header.html" %}
  <div>
    {% block content %}{% endblock %}
  </div>
  {% include "footer.html" %}
</body>
</html>
```

Copy code

child.html

```
html

{% extends "base.html" %}

{% block title %}Home Page{% endblock %}

{% block content %}
  <h1>Welcome to the Home Page</h1>
  <p>This content comes from the child template.</p>
{% endblock %}
```

Copy code

- **Flow:** Child template inherits **base.html**, overrides blocks like **title** and **content**.

◆ 7. Template Filters (Bonus)

- Modify variables in templates. Syntax: `{{ variable|filter }}`
- Common filters:
 - `{{ name|lower }}` → convert to lowercase
 - `{{ name|upper }}` → convert to uppercase
 - `{{ value|length }}` → get length of list/string
 - `{{ date|date:"D M Y" }}` → format date

✓ Viva One-liners

1. "A template is an HTML file with dynamic content using Django Template Language."
2. "Template tags `{% %}` add logic, loops, and inheritance in templates."
3. "Variables `{{ }}` display data from views/models."
4. "Template inheritance allows child templates to reuse base layouts."
5. "Filters modify data in templates, e.g., `upper`, `lower`, `length`."



who is parent and who is child

Great question 👍

In Django templates, the parent and child relationship comes from template inheritance using `{% extends %}` and `{% block %}`.

🔧 Parent vs Child Templates

👤 Parent Template → `base.html`

- It's the master layout.
- Defines the common structure (header, footer, CSS, JS, navbar, etc.).
- Uses `{% block %}` as placeholders for children to fill.

👉 Example (Parent):

```
django 📄 Copy code

<!-- base.html -->
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <header>Header for all pages</header>

    <!-- Block where child content will go -->
    {% block content %}{% endblock %}

    <footer>Footer for all pages</footer>
  </body>
</html>
```

👤 Child Template → `index.html`, `about.html`, etc.

- These are individual pages.
- They say: "I want to use `base.html` as parent."
- They extend the parent and fill the blocks.

👉 Example (Child):

```
django 📄 Copy code
```

👉 Example (Child):

django

📄 Copy code

```
<!-- index.html -->
{% extends "base.html" %}

{% block content %}
    <h2>Welcome to Home Page</h2>
    <p>This is the child template content.</p>
{% endblock %}
```

🔗 How They Work Together

1. Child template (`index.html`) says → "I extend `base.html` ".
2. Django looks into parent (`base.html`).
3. Wherever it finds `{% block content %}` , it replaces it with the child's content.

📄 Final Output in Browser

html

📄 Copy code

```
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <header>Header for all pages</header>

    <h2>Welcome to Home Page</h2>
    <p>This is the child template content.</p>

    <footer>Footer for all pages</footer>
  </body>
</html>
```

✅ Summary

- Parent = `base.html` (layout, structure, reusability).
- Child = `index.html` , `about.html` , etc. (extends parent, fills blocks).
- Relationship:
 - Parent provides the **skeleton**.
 - Child provides the **page-specific content**.