

What is a Web Framework?

A web framework, or web application framework (WAF), is a software framework that helps developers build web applications. It provides a standard way to build and deploy web applications on the World Wide Web. A web framework aims to make it easier for developers to build complex web applications by providing a structured approach to application development. Web app frameworks for both front-end and back-end development promote best practices and standardized processes, allowing developers to improve code quality and maintainability.

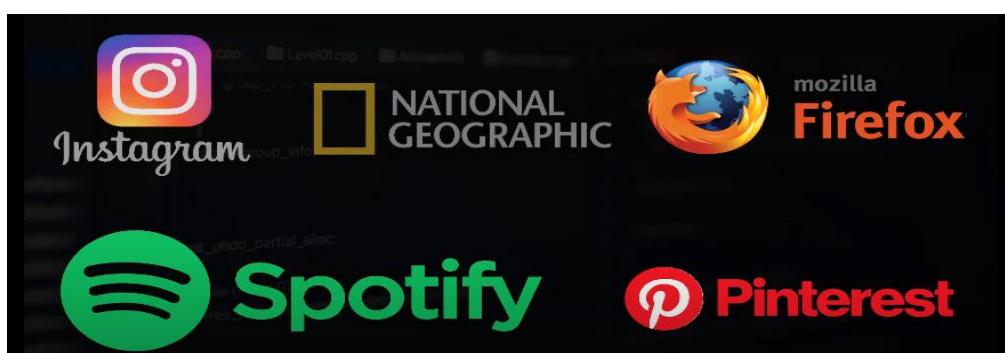
What is a Django?

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel.

Django is a web framework written in [Python](#). A web framework is a software that supports the development of dynamic websites, applications, and services. It provides a set of tools and functionalities that solves many common problems associated with web development, such as security features, database access, sessions, template processing, URL routing, internationalization, localization, and much more.

Why Django Web Framework?

First of all, it is a Python web framework, which means that you can benefit from a wide range of open-source libraries out there. This popular web framework also offers a standalone [web-server](#) for development and testing, caching, and middleware system. Moreover, it provides the ORM (object-relational mapper) library, the template engine, form processing, and an interface with Python's unit testing tools.



Key features of Django include:

- **Object-Relational Mapper (ORM):** Provides an interface for interacting with databases using Python code.
- **Routing:** Defines URL patterns and maps them to specific views.
- **Templating engine:** Enables dynamic HTML generation.
- **Forms:** Simplifies the process of creating and validating user input forms.
- **Admin interface:** Automatically generates a user-friendly interface for managing data.
- **Security features:** Protects against common web vulnerabilities.

Advantages of Django:-

- Rapid Development.
- Secure.
- Scalable.
- Fully loaded.
- Versatile.
- Open Source.
- Vast and Supported Community

Rapid Development

It takes less time to build web application. The project implementation phase is a very time taken but Django creates it rapidly.

Secure

Django takes security seriously and helps developers to avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery etc. Its user authentication system provides a secure way to manage user accounts and passwords.

Scalable

Django is scalable in nature and has ability to quickly and flexibly switch from small to large scale application project.

Fully loaded

Django includes various helping task modules and libraries which can be used to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds etc.

Versatile

Django is versatile in nature which allows it to build applications for different-different domains. Now a days, Companies are using Django to build various types of applications like: content management systems, social networks sites or scientific computing platforms etc.

Open Source

Django is an open source web application framework. It is publicly available without cost. It can be downloaded with source code from the public repository. Open source reduces the total cost of the application development.

Vast and Supported Community

Django is an one of the most popular web framework. It has widely supportive community and channels to share and connect.

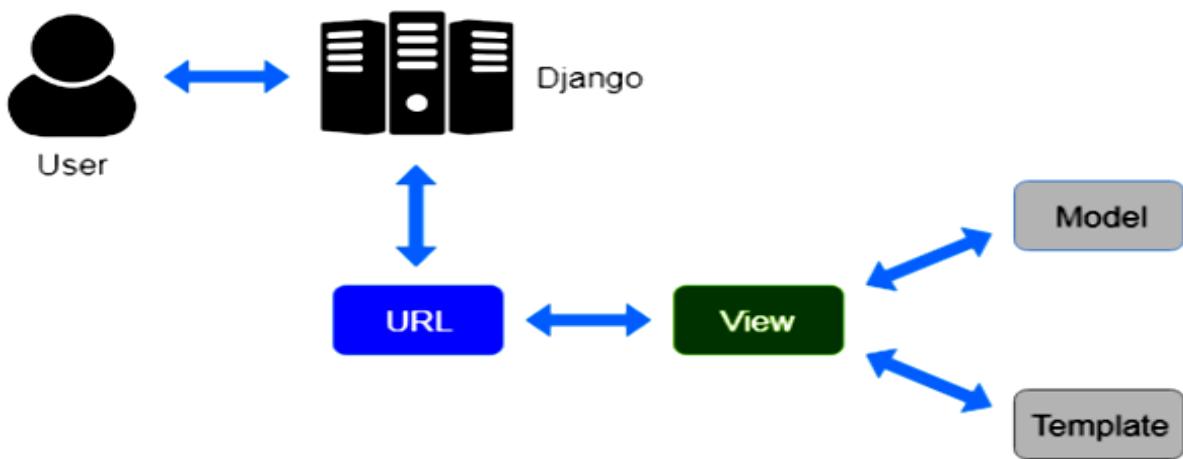
Disadvantages of using Django:

- 1. Higher memory usage:-** Django is known to use more memory than other frameworks, such as Flask or Pyramid. This can be an issue for projects with high traffic or memory constraints.
- 2. Monolithic structure:-** Django follows a monolithic structure, which means that all the components of an application are tightly coupled. This can make it difficult to customize or extend an application, and it can also lead to performance problems for large applications.
- 3. Not suitable for smaller projects:-** Django is a large and complex framework, which can make it overkill for smaller projects. There are other frameworks, such as Flask or Pyramid, that are better suited for smaller projects.
- 4. Limited support for non-relational databases:-** Django primarily supports relational databases, such as PostgreSQL and MySQL. While there is some support for non-relational databases, such as MongoDB, it is not as comprehensive as the support for relational databases.

Django works in MVT-pattern:-

The MVT (Model View Template) is a software design pattern. It is a collection of three important components Model View and Template. The Model helps to handle database. It is a data access layer which handles the data.

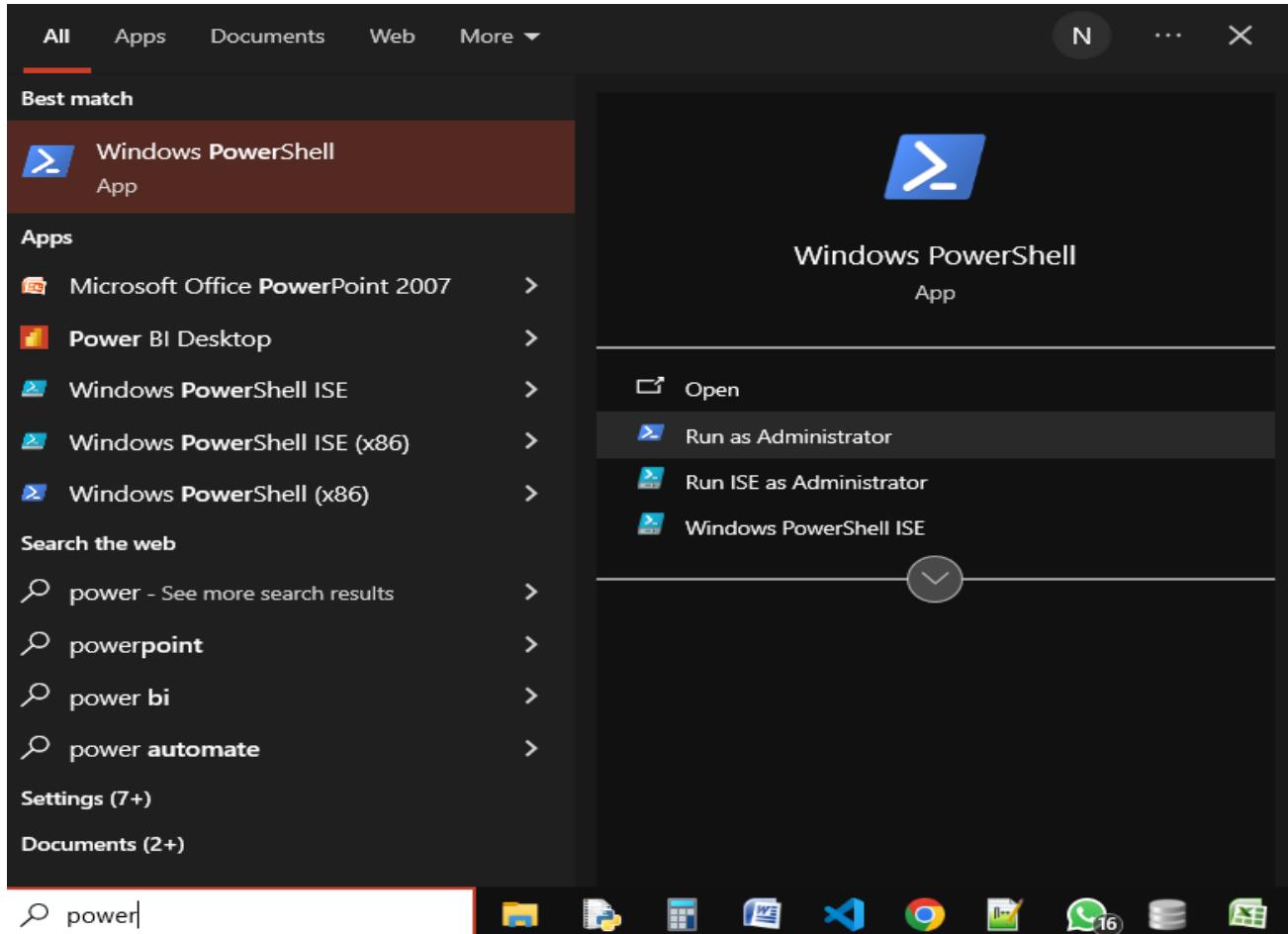
The Template is a presentation layer which handles User Interface part completely. The View is used to execute the business logic and interact with a model to carry data and renders a template.



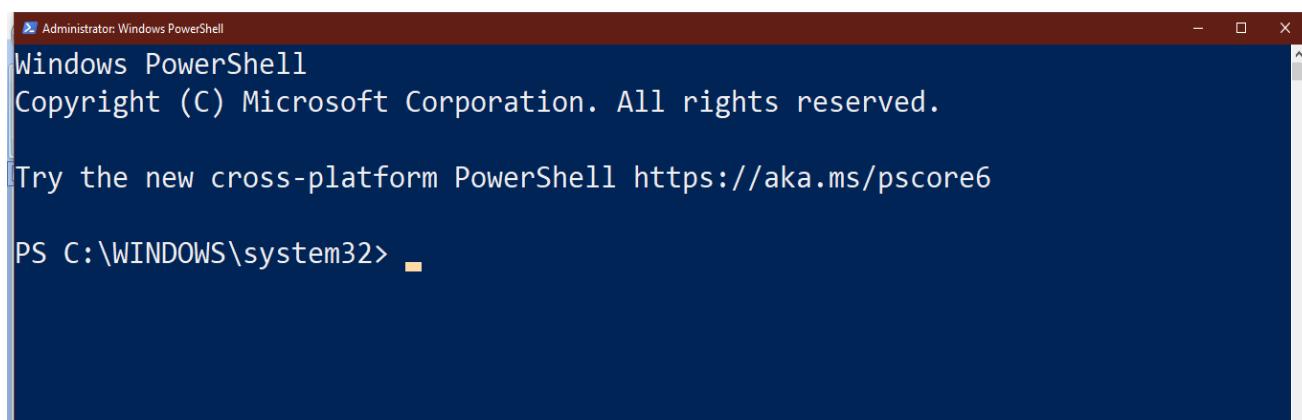
One of the most significant differences lies in how user interactions are managed. In MVT, the Template is mainly for presentation, while the View handles data and user interactions. In MVC, the Controller is responsible for handling user interactions and updating the Model, which in turn updates the View.

Enable Scripting from windows power-shell

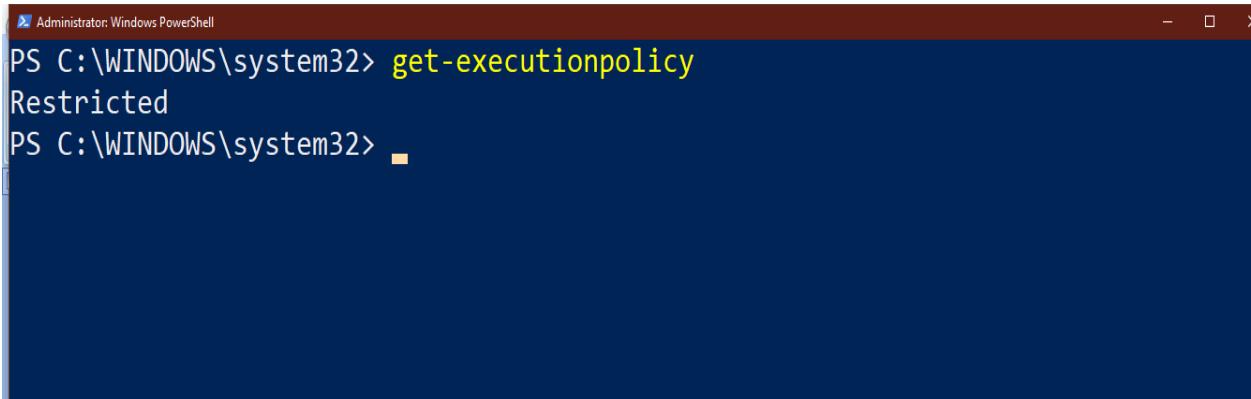
Step 1:- Open Windows powershell



Step 2:- Click on Run as Administrator

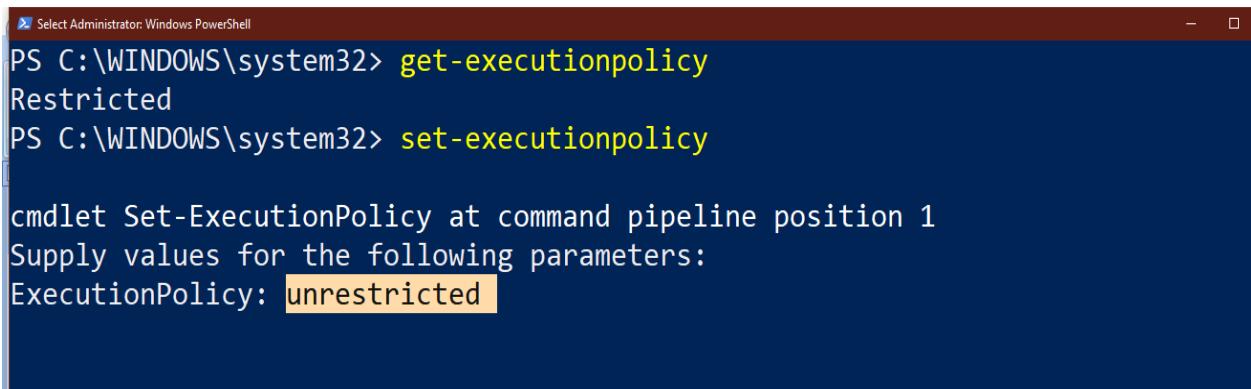


Step 3 :- Check current executionpolicy



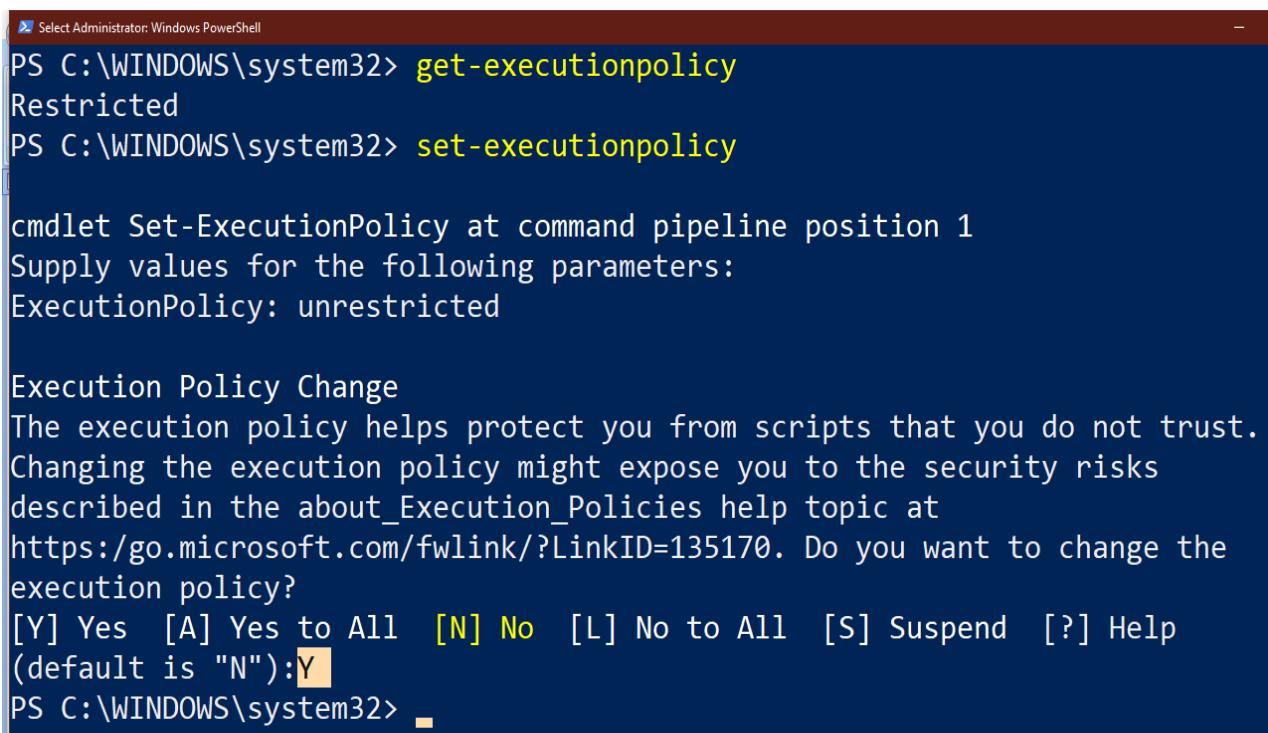
```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> get-executionpolicy
Restricted
PS C:\WINDOWS\system32> ■
```

Step 4:- Change current execution policy



```
Select Administrator: Windows PowerShell
PS C:\WINDOWS\system32> get-executionpolicy
Restricted
PS C:\WINDOWS\system32> set-executionpolicy

cmdlet Set-ExecutionPolicy at command pipeline position 1
Supply values for the following parameters:
ExecutionPolicy: unrestricted
```

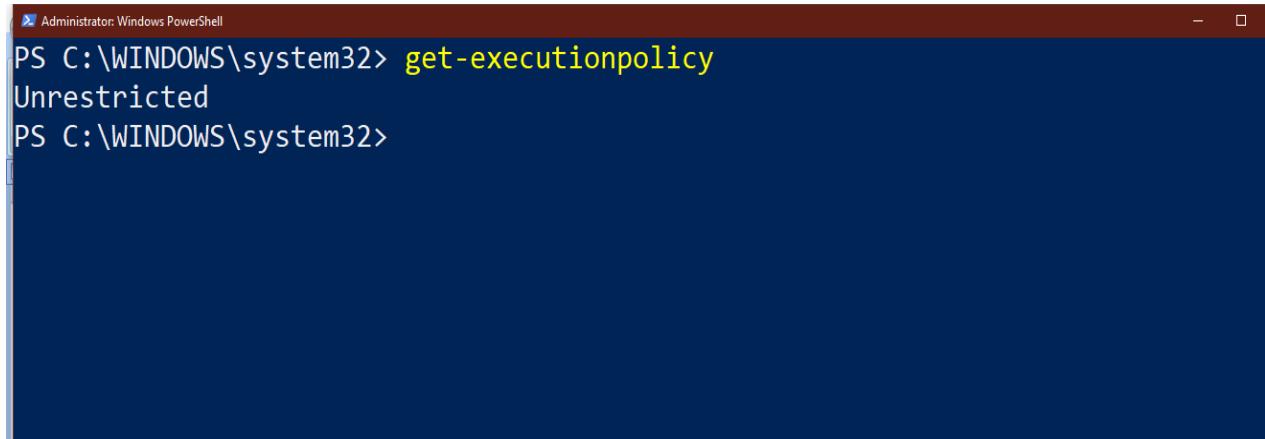


```
Select Administrator: Windows PowerShell
PS C:\WINDOWS\system32> get-executionpolicy
Restricted
PS C:\WINDOWS\system32> set-executionpolicy

cmdlet Set-ExecutionPolicy at command pipeline position 1
Supply values for the following parameters:
ExecutionPolicy: unrestricted

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust.
Changing the execution policy might expose you to the security risks
described in the about_Execution_Policies help topic at
https://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the
execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "N"):Y
PS C:\WINDOWS\system32> ■
```

Step 5:- Again check current executionpolicy



```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> get-executionpolicy
Unrestricted
PS C:\WINDOWS\system32>
```

Virtual Environment:--

What is a Virtual Environment?

A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated Python virtual environments for them. This is one of the most important tools that most Python developers use.

Why do we need a virtual environment?

Imagine a scenario where you are working on two web-based Python projects one of them uses Django 4.0 and the other uses Django 5.0 (check for the latest Django versions and so on). In such situations, we need to create a virtual environment in Python that can be really useful to maintain the dependencies of both projects.

How to create **Virtual Environment:**

Step 1:- Open cmd

Step 2:- python -m venv env / py -m venv my_env

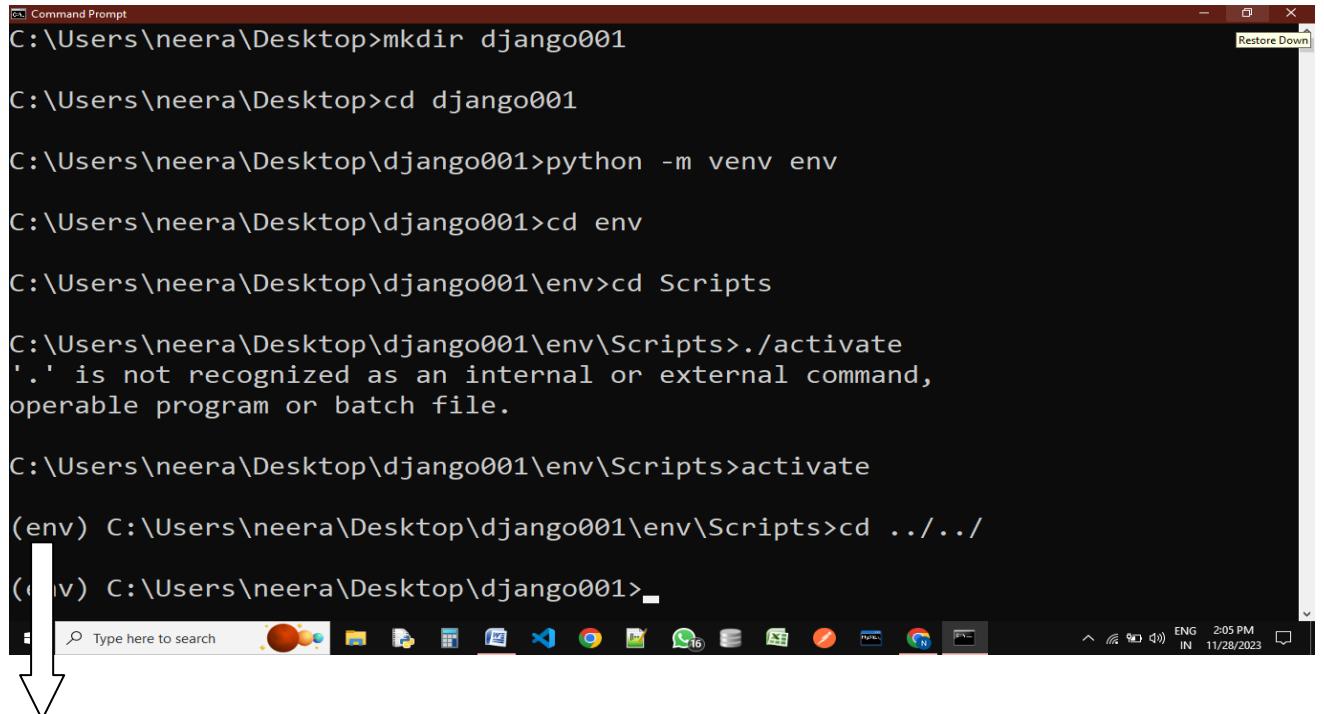
(In Python, the **-m** flag is used to run a module as a script. When you execute **python -m <module_name>**, Python will execute the specified module as the main program.)

Step 3:- cd env/Scripts/

Step 4:- ./activate

Step 5:- cd ../../

Now, we create and activate virtual environment.



```
C:\Users\neera\Desktop>mkdir django001
C:\Users\neera\Desktop>cd django001
C:\Users\neera\Desktop\django001>python -m venv env
C:\Users\neera\Desktop\django001>cd env
C:\Users\neera\Desktop\django001\env>cd Scripts
C:\Users\neera\Desktop\django001\env\Scripts>./activate
'.' is not recognized as an internal or external command,
operable program or batch file.
C:\Users\neera\Desktop\django001\env\Scripts>activate
(env) C:\Users\neera\Desktop\django001\env\Scripts>cd ../../
(env) C:\Users\neera\Desktop\django001>
```

A large downward-pointing arrow is positioned to the left of the command prompt window, pointing towards the text.

This indicates that you create and activate virtual environment.

How to Install Django

Prerequisites for Installing Django

Check if Python is installed in the system by using the following command:

Step 1:- Open cmd prompt.

Step 2:- check python is installed or not



```
Microsoft Windows [Version 10.0.19045.3693]
(c) Microsoft Corporation. All rights reserved.

C:\Users\neera>python --version
Python 3.10.0
```

Step 3: If Python is not installed in your system, then install it through this <https://www.python.org/> web page.

Step 4:- After installing python again check python installed or not through cmd.

Step 5:- In cmd **pip install django.(pip install django==4.2.5)**

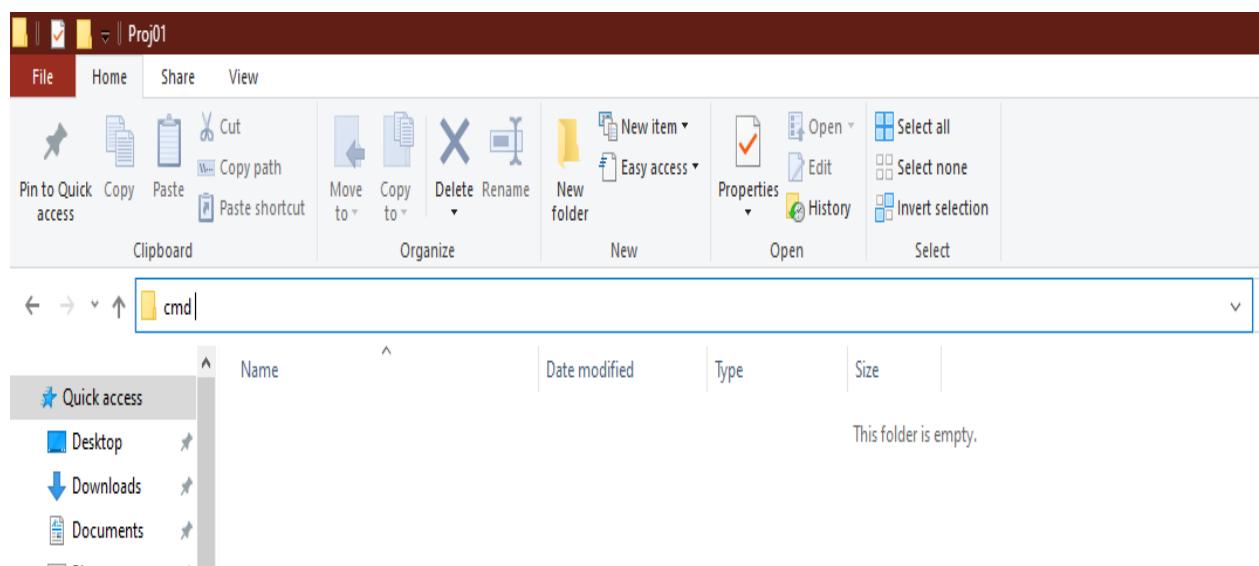
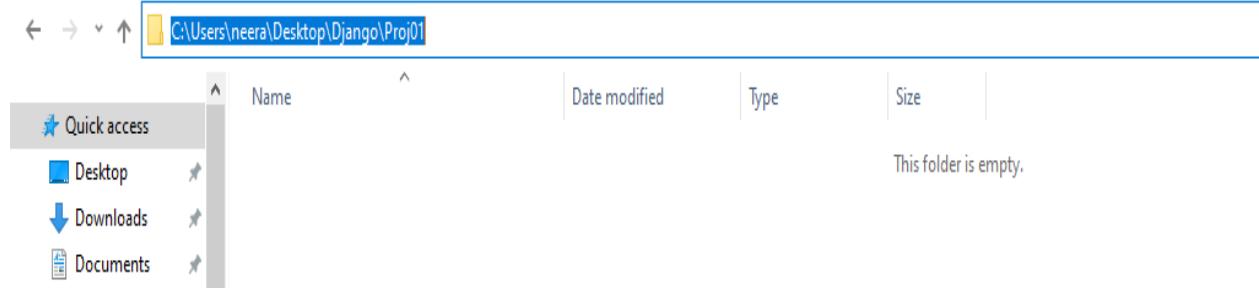
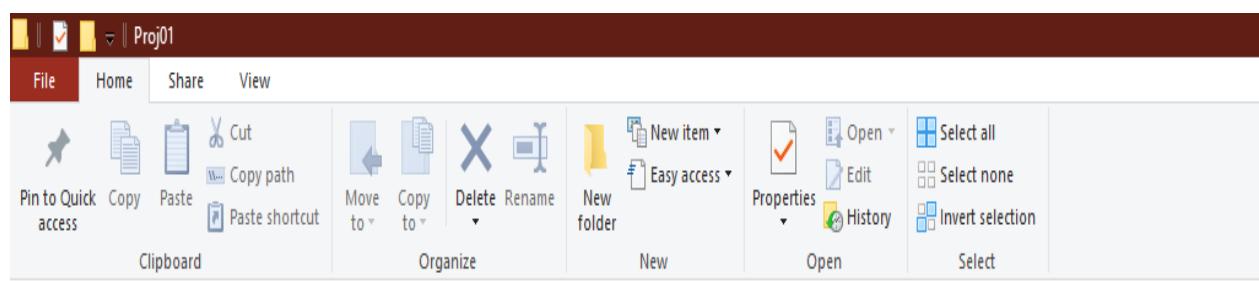
Step 6:- check django install or not through **Django-admin --version** in cmd prompt.

```
Command Prompt
Microsoft Windows [Version 10.0.19045.3693]
(c) Microsoft Corporation. All rights reserved.

C:\Users\neera>python --version
Python 3.10.0

C:\Users\neera>Django-admin --version
4.2.5
```

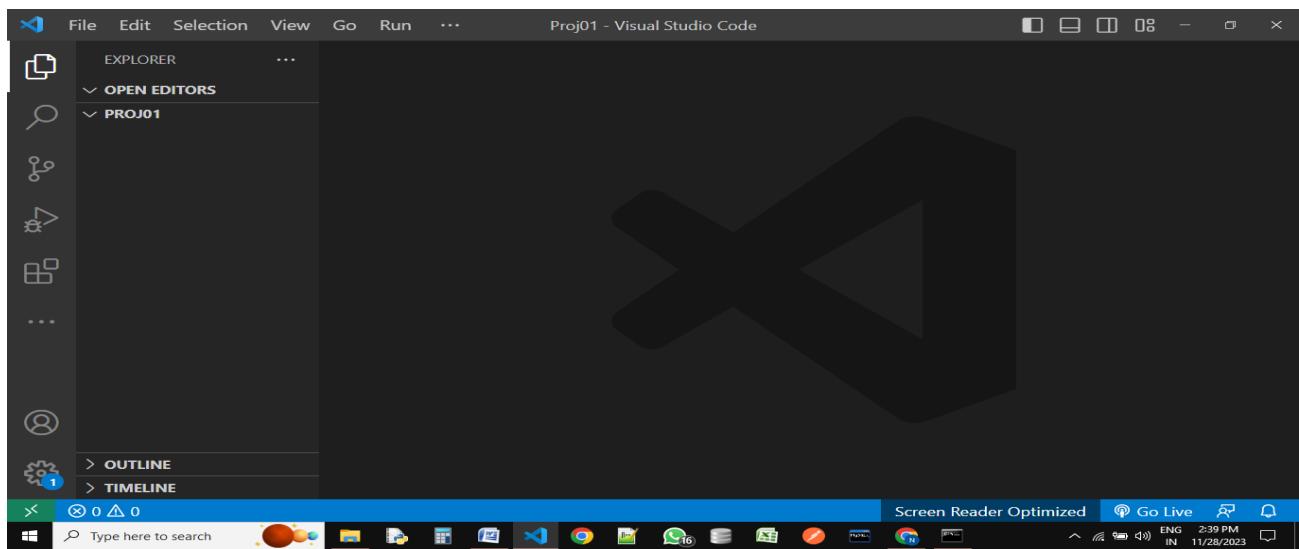
Create first Django Project:



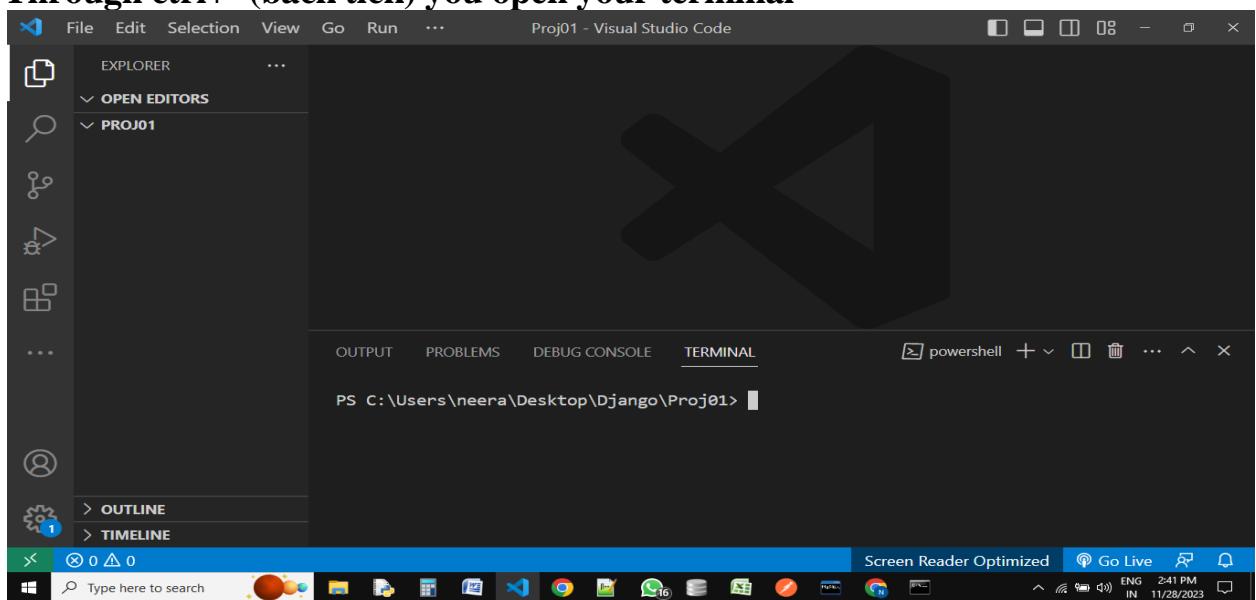
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.3693]
(c) Microsoft Corporation. All rights reserved.

C:\Users\neera\Desktop\Django\Proj01>code .
```

After that it directly open VS-Code:

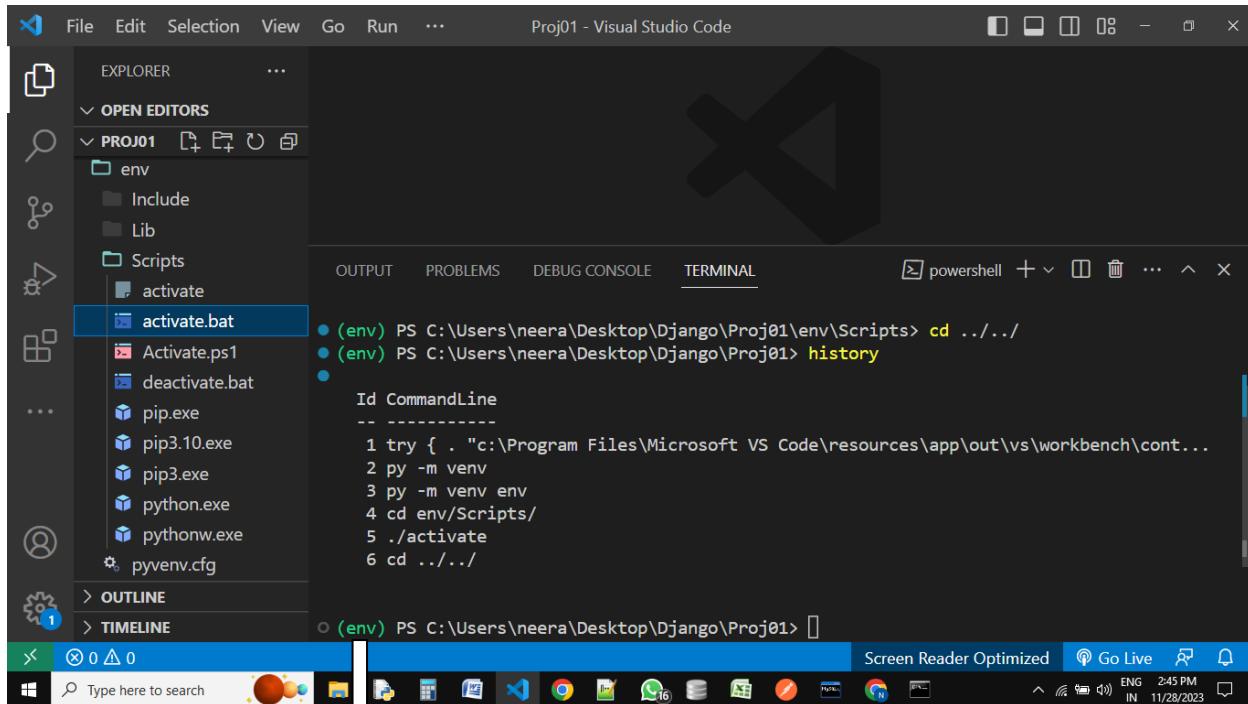


Through **ctrl+`** (back tick) you open your terminal



Create and activate virtual environment through following steps:

Step 1:- py -m venv env -----enter
Step 2:- cd env/Scripts/ -----enter
Step 3:- ./activate -----enter
Step 4:- cd ../../ -----enter



The screenshot shows the Visual Studio Code interface with the title "Proj01 - Visual Studio Code". The left sidebar displays the file structure of "PROJ01", including a "env" folder containing "Include", "Lib", "Scripts", and "activate" files. The "activate.bat" file is selected. The main area is the "TERMINAL" tab, which shows the following command history:

```
(env) PS C:\Users\neera\Desktop\ Django\Proj01\env\Scripts> cd ../../
(env) PS C:\Users\neera\Desktop\ Django\Proj01> history
Id CommandLine
-----
1 try { . "c:\Program Files\Microsoft VS Code\resources\app\out\vs\workbench\cont...
2 py -m venv
3 py -m venv env
4 cd env/Scripts/
5 ./activate
6 cd ../../

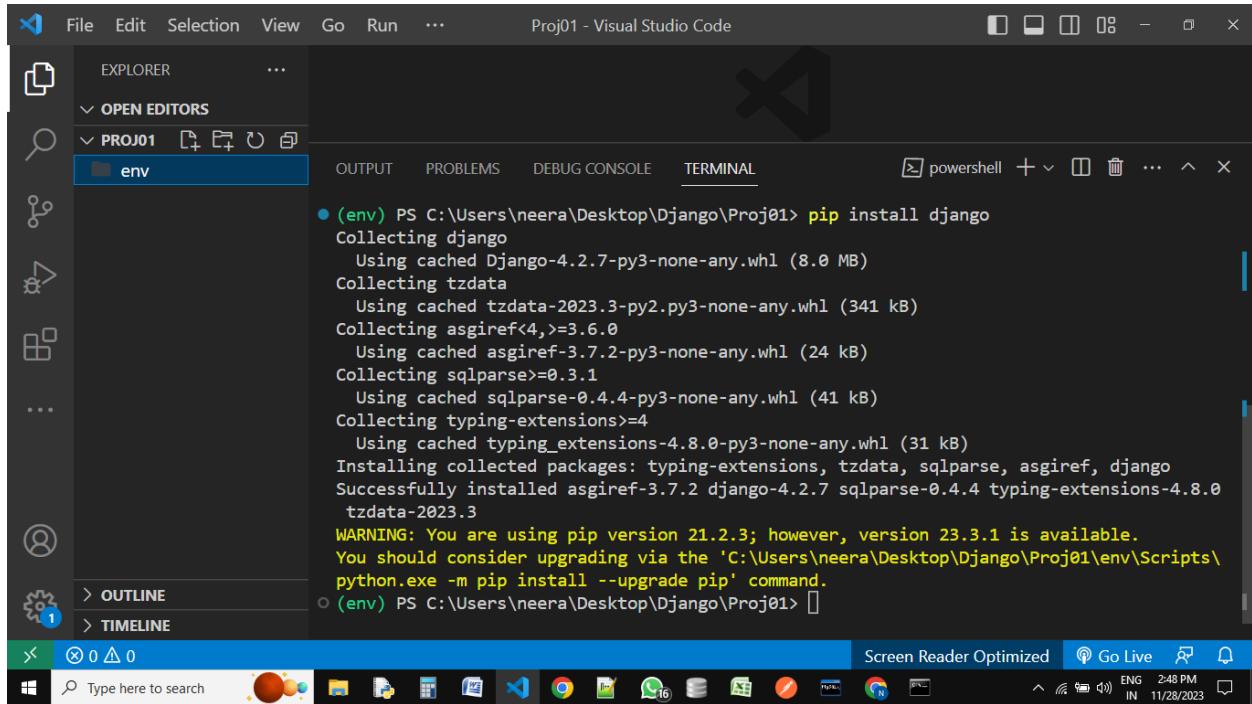
(env) PS C:\Users\neera\Desktop\ Django\Proj01>
```

The terminal output indicates that the virtual environment has been successfully created and activated.

This indicates that your virtual env is created and successfully activated.

Install django and create new project

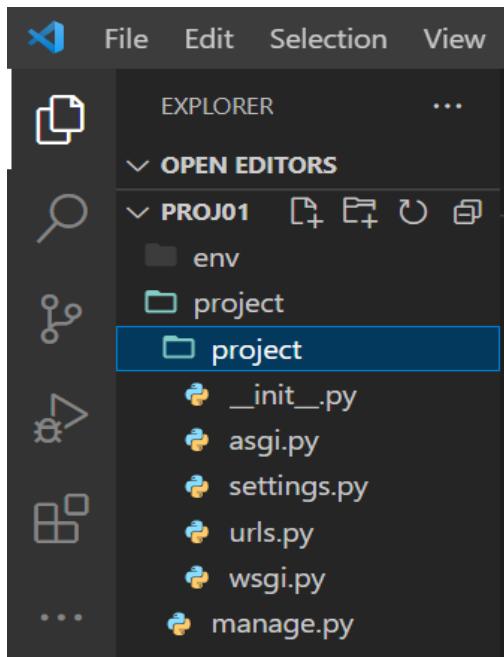
Step 1:- pip install django



The screenshot shows the Visual Studio Code interface with the title "Proj01 - Visual Studio Code". The terminal tab is active, displaying the command "pip install django" and its execution. The output shows the download and installation of various packages, including django version 4.2.7, tzdata, asgiref, sqlparse, typing-extensions, and others. A warning message at the end indicates that pip version 21.2.3 is being used, while version 23.3.1 is available.

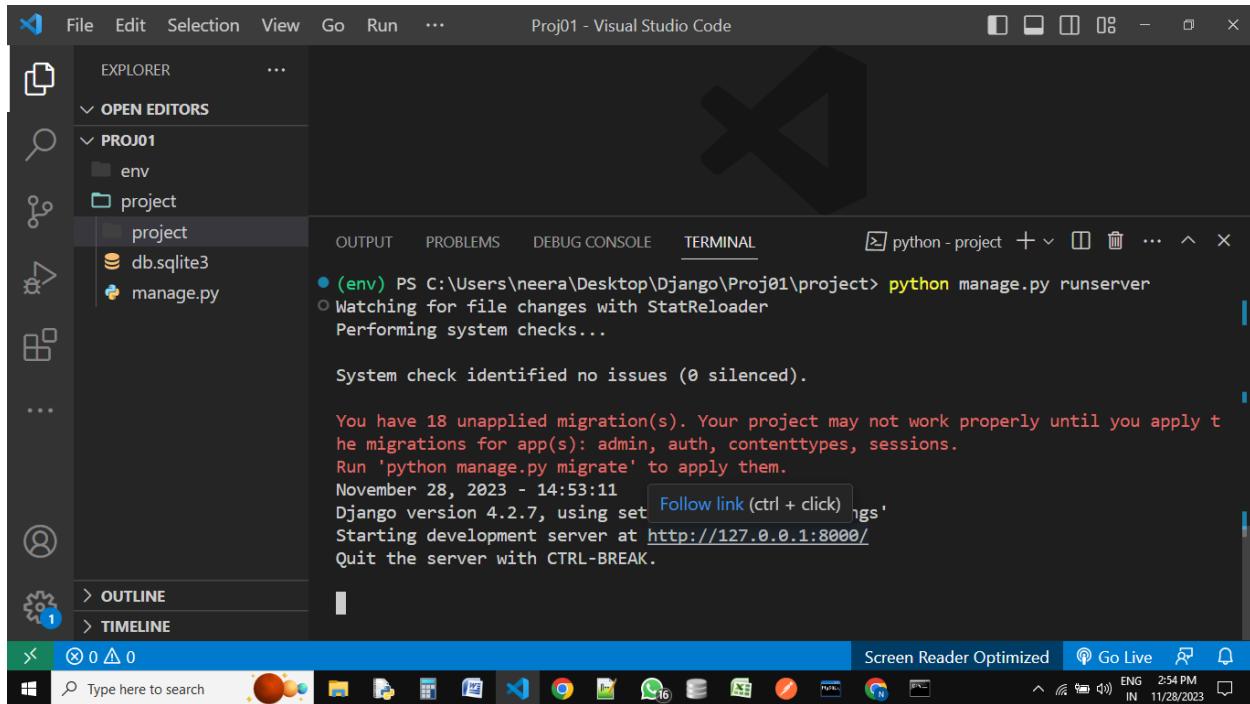
```
(env) PS C:\Users\neera\Desktop\Django\Proj01> pip install django
Collecting django
  Using cached Django-4.2.7-py3-none-any.whl (8.0 MB)
Collecting tzdata
  Using cached tzdata-2023.3-py2.py3-none-any.whl (341 kB)
Collecting asgiref<4,>=3.6.0
  Using cached asgiref-3.7.2-py3-none-any.whl (24 kB)
Collecting sqlparse>=0.3.1
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Collecting typing-extensions>=4
  Using cached typing_extensions-4.8.0-py3-none-any.whl (31 kB)
Installing collected packages: typing-extensions, tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.7.2 django-4.2.7 sqlparse-0.4.4 typing-extensions-4.8.0
tzdata-2023.3
WARNING: You are using pip version 21.2.3; however, version 23.3.1 is available.
You should consider upgrading via the 'C:\Users\neera\Desktop\Django\Proj01\env\Scripts\python.exe -m pip install --upgrade pip' command.
(env) PS C:\Users\neera\Desktop\Django\Proj01>
```

Step 2:- Django-admin startproject project (Creating new django project)—
Inside Django project we found one inner project folder that contains:--



Step 3:- cd project

Step 4:- python manage.py runserver (Running the Django Project)



The screenshot shows the Visual Studio Code interface with the terminal tab selected. The terminal window displays the command 'python manage.py runserver' being executed in an environment named 'env'. The output shows system checks, migration status, and the start of the development server at 'http://127.0.0.1:8000/'.

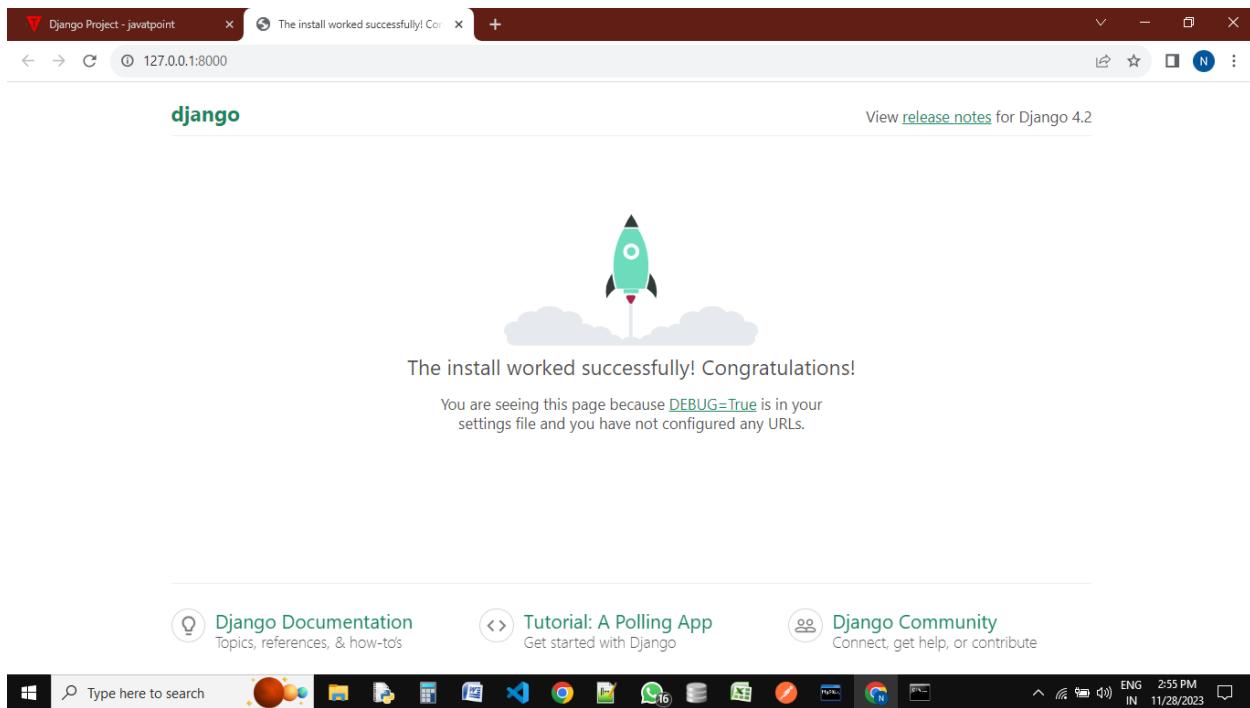
```
(env) PS C:\Users\neera\Desktop\ Django\Proj01\project> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

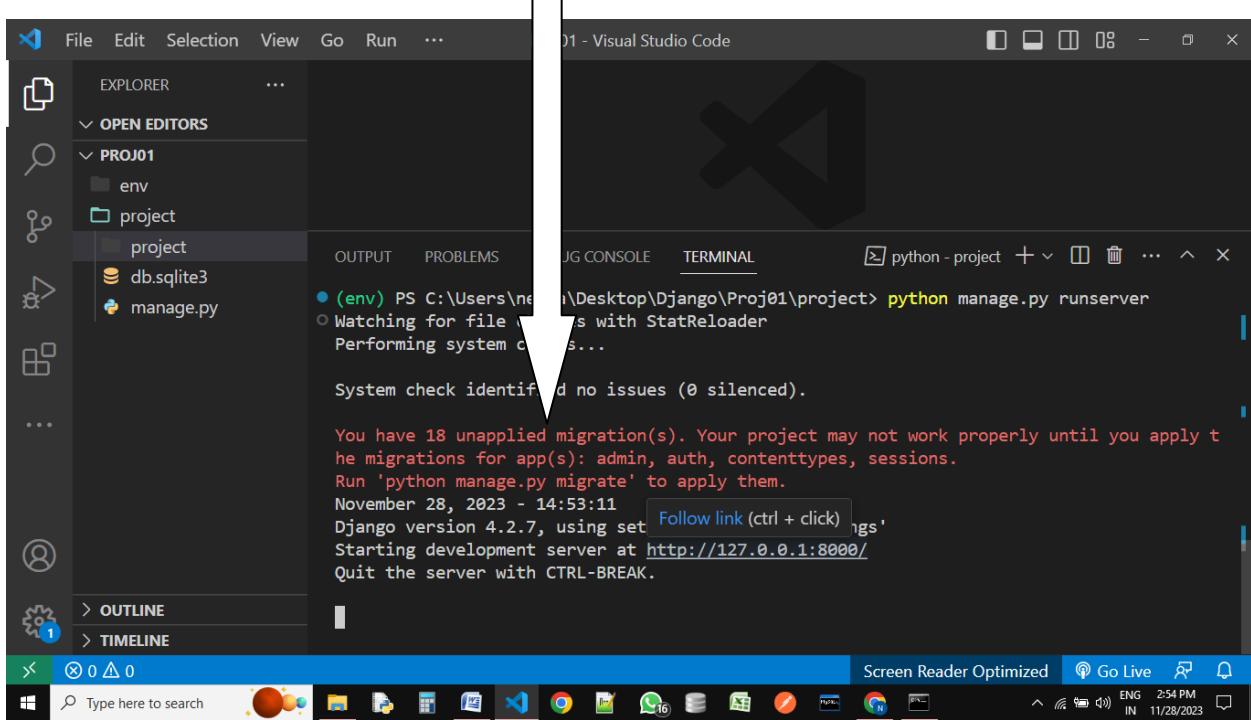
You have 18 unapplied migration(s). Your project may not work properly until you apply t
he migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

November 28, 2023 - 14:53:11
Django version 4.2.7, using settings 'migrations'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Step 5:- click Follow link



Remove Red colour error that comming from runserver O/P:--



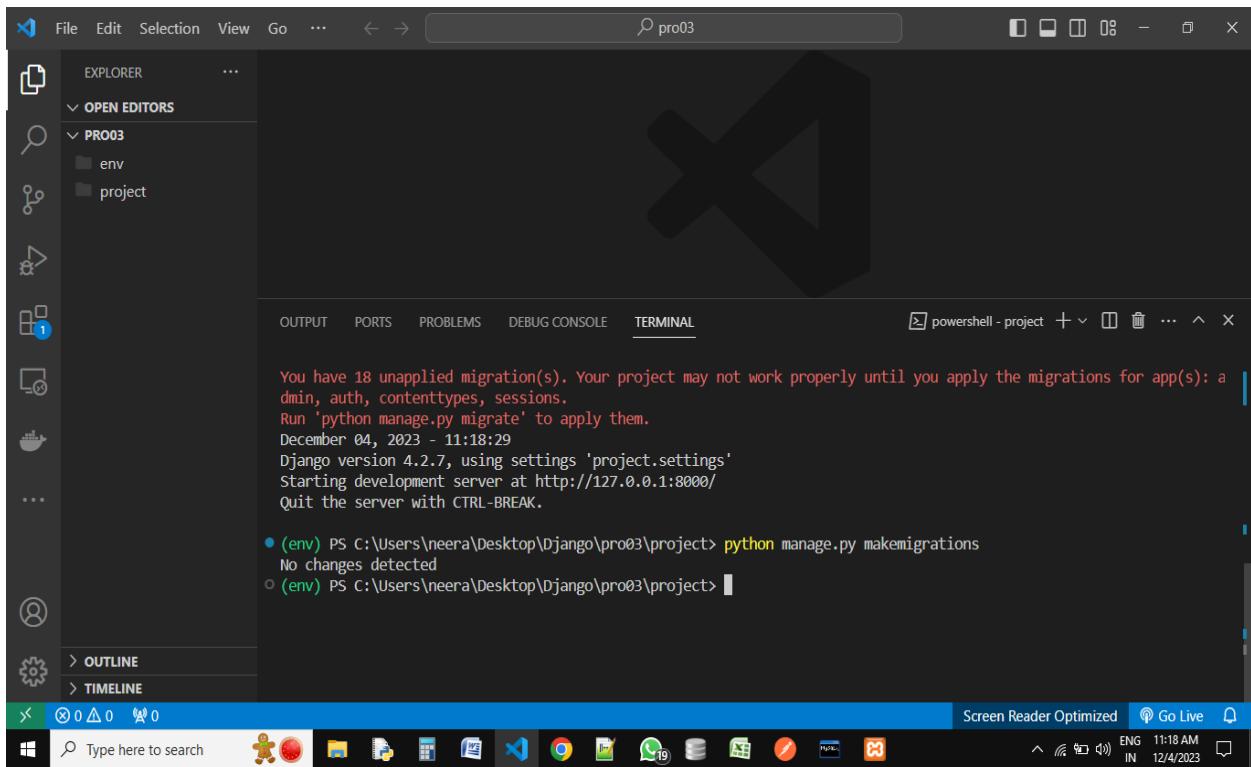
Screenshot of Visual Studio Code showing the terminal output of 'python manage.py runserver'. The terminal window shows the command being run, the system check results, and the start of the development server.

```
(env) PS C:\Users\neera\Desktop\ Django\Proj01\project> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

November 28, 2023 - 14:53:11
Django version 4.2.7, using settings 'project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Step 6:- python manage.py makemigrations (it responsible to create query file)



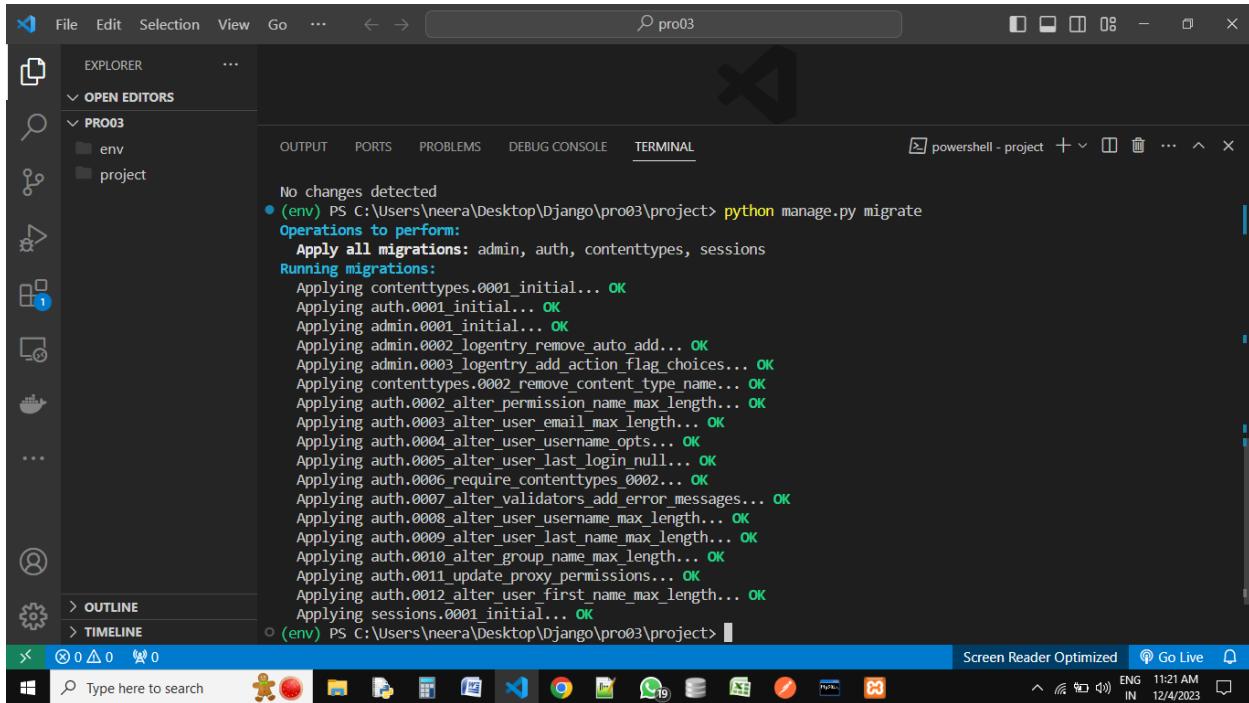
Screenshot of Visual Studio Code showing the terminal output of 'python manage.py makemigrations'. The terminal window shows the command being run, the detection of no changes, and the start of the development server.

```
You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

December 04, 2023 - 11:18:29
Django version 4.2.7, using settings 'project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

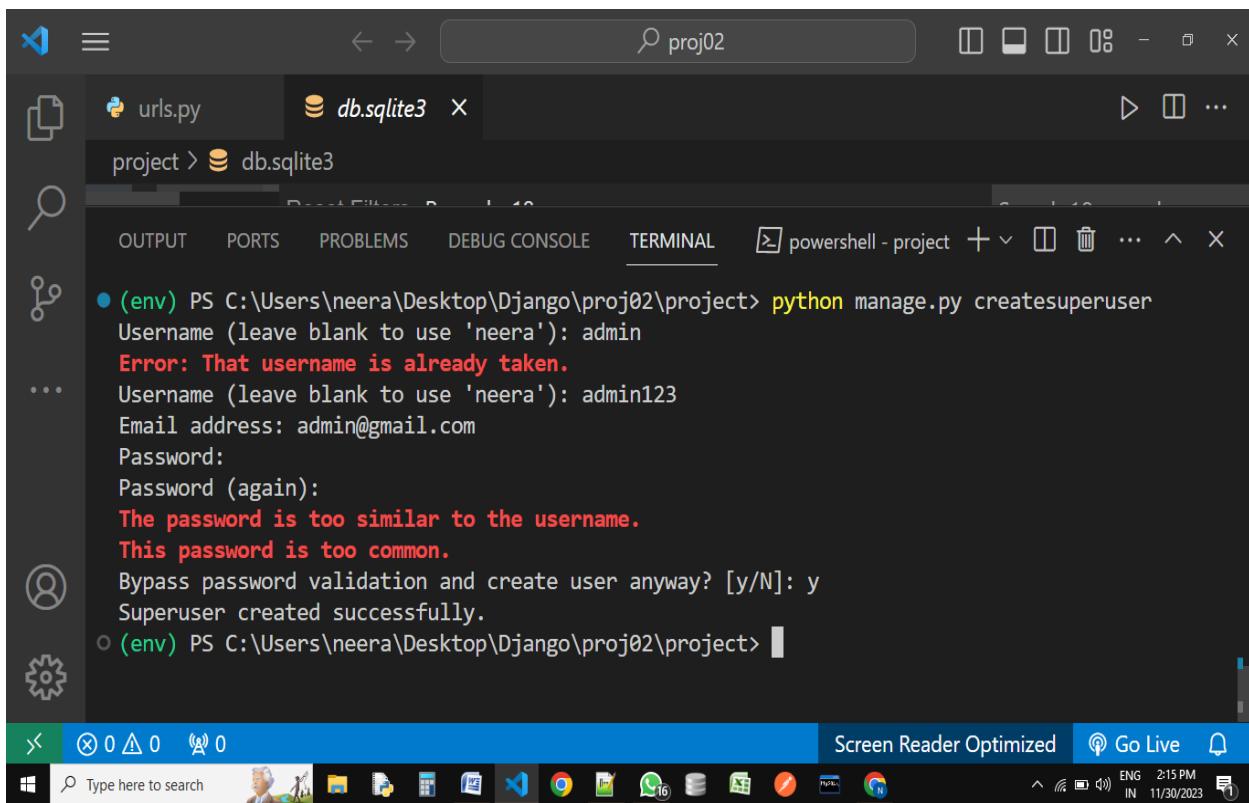
● (env) PS C:\Users\neera\Desktop\ Django\pro03\project> python manage.py makemigrations
No changes detected
○ (env) PS C:\Users\neera\Desktop\ Django\pro03\project>
```

Step 7:- python manage.py migrate (it will generate table in database)



```
No changes detected
● (env) PS C:\Users\neera\Desktop\ Django\pro03\project> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
○ (env) PS C:\Users\neera\Desktop\ Django\pro03\project>
```

Step 8:- python manage.py createsuperuser

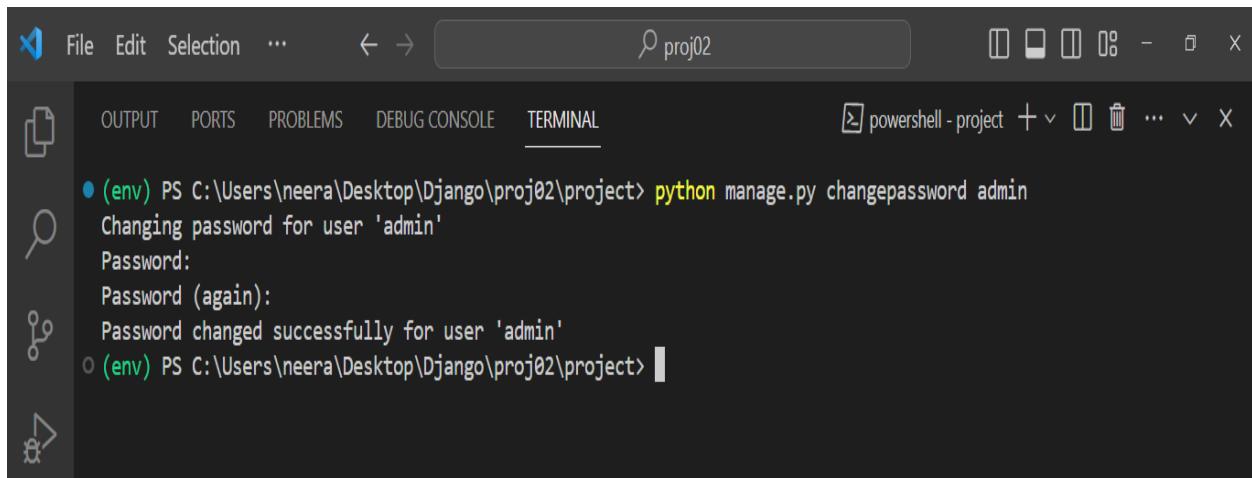


```
● (env) PS C:\Users\neera\Desktop\ Django\proj02\project> python manage.py createsuperuser
Username (leave blank to use 'neera'): admin
Error: That username is already taken.
Username (leave blank to use 'neera'): admin123
Email address: admin@gmail.com
Password:
Password (again):
The password is too similar to the username.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
○ (env) PS C:\Users\neera\Desktop\ Django\proj02\project>
```

Fields:---

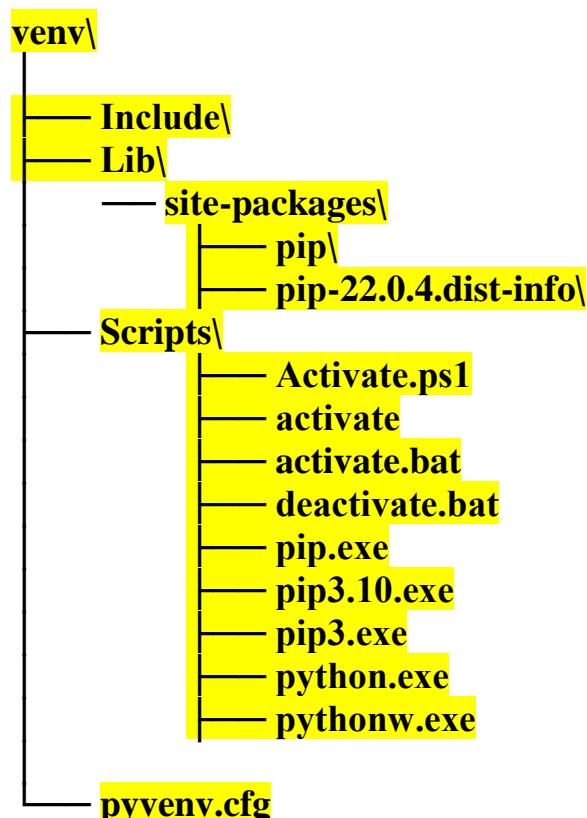
Username *	required
Email address	Optional
Password *	required (always in hidden form)
Password(again) *	required (always in hidden form)

Step 9:- python manage.py changepassword user_name*)



```
(env) PS C:\Users\neera\Desktop\ Django\proj02\project> python manage.py changepassword admin
Changing password for user 'admin'
Password:
Password (again):
Password changed successfully for user 'admin'
o (env) PS C:\Users\neera\Desktop\ Django\proj02\project>
```

Virtual Environment Folder structure:--



- **Include** is an initially empty folder that Python uses to include C header files for packages you might install that depend on C extensions.
- **Lib** contains the site-packages\ folder, which is one of the main reasons for creating your virtual environment. This folder is where you'll install external packages that you want to use within your virtual environment. By default, your virtual environment comes preinstalled with two dependencies, pip and set up tools.
- **Scripts** contains the executable files of your virtual environment. Most important files are the Python interpreter (python.exe), the pip executable (pip.exe), and the activation script for your virtual environment, which comes in a couple of different flavors to allow you to work with different shells.
- **pyvenv.cfg** is a crucial file for your virtual environment. It contains only a couple of key-value pairs that Python uses to set variables in the sys module that determine which Python interpreter and which site-packages directory the current Python session will use.

Project Folder structure:-- (Django-admin startproject project)

project(outer project folder structure)

```

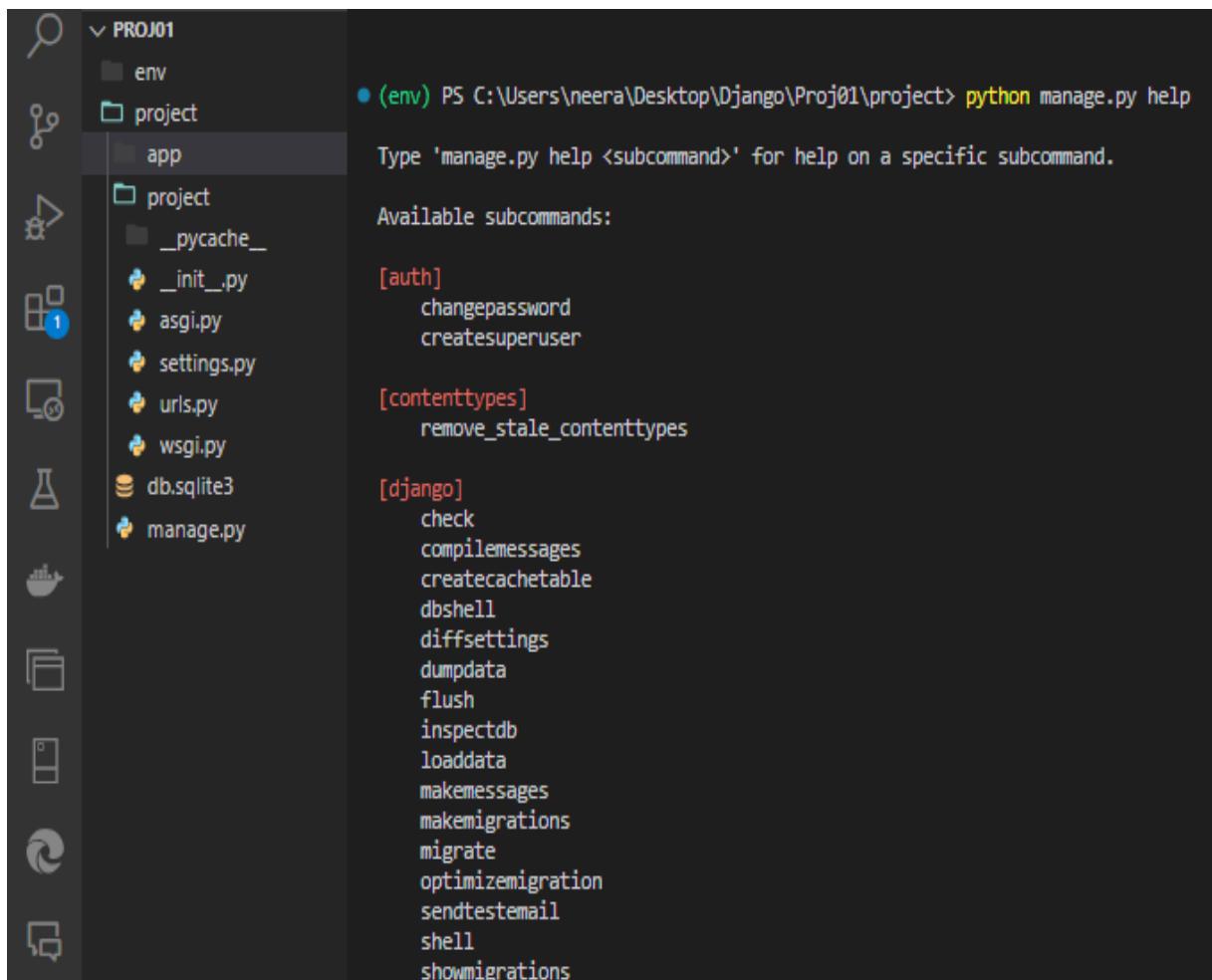
---- project/(inner project folder structure)
      |---- __pycache__/
      |---- __init__.py
      |---- asgi.py
      |---- settings.py
      |---- urls.py
      |---- wsgi.py
---- db.sqlite3
---- manage.py

```

1. **manage.py**: django-admin is **Django's command-line utility tools for administrative tasks**. Or It is a command-line utility which allows us to interact with the project in various ways and also used to manage an application.
2. **__init__.py** : It is an empty file that tells to the Python that this directory should be considered as a Python package.
3. **settings.py** : This file is used to configure application settings such as database connection, static files linking etc.

4. **urls.py** : This file contains the listed URLs of the application. In this file, we can mention the URLs and corresponding actions to perform the task and display the view.
5. **wsgi.py(Web Server Gateway Interface)** : Web Server Gateway Interface (WSGI) is a mediator responsible for conveying communication between a web server and a Python web application. It is an entry-point for WSGI-compatible web servers to serve Django project.
6. **asgi.py(Asynchronous Server Gateway Interface)** : Unlike WSGI, ASGI allows multiple, asynchronous events per application. Plus, ASGI supports both sync and async apps. You can migrate your old, synchronous WSGI web apps to ASGI, as well as use ASGI to build new, asynchronous web apps.

With the help of manage.py we can perform specific task with the help of below mention commands:-----



```
● (env) PS C:\Users\neera\Desktop\ Django\Proj01\project> python manage.py help
Type 'manage.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[auth]
    changepassword
    createsuperuser

[contenttypes]
    remove_stale_contenttypes

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    optimizemigration
    sendtestemail
    shell
    showmigrations
```

The screenshot shows a terminal window with the following details:

- Project Structure:** The left sidebar shows a project structure with a folder named "PROJ01" containing "env" and "project". Inside "project", there is a folder "app" and several files: "project", "__pycache__", "__init__.py", "asgi.py", "settings.py", "urls.py", and "wsgi.py".
- Terminal Command:** The command entered is "python manage.py help".
- Output:**
 - A message: "Type 'manage.py help <subcommand>' for help on a specific subcommand."
 - A list of "Available subcommands" grouped by category:
 - [auth]**: changepassword, createsuperuser
 - [contenttypes]**: remove_stale_contenttypes
 - [django]**: check, compilemessages, createcachetable, dbshell, diffsettings, dumpdata, flush, inspectdb, loaddata, makemessages, makemigrations, migrate, optimizemigration, sendtestemail, shell, showmigrations

```
sqlflush
sqlmigrate
sqlsequencereset
squashmigrations
startapp
startproject
test
testserver

[sessions]
clearsessions
```

With the help of Django-admin we use below mention commands to perform specific task.

Project vs app in django:---

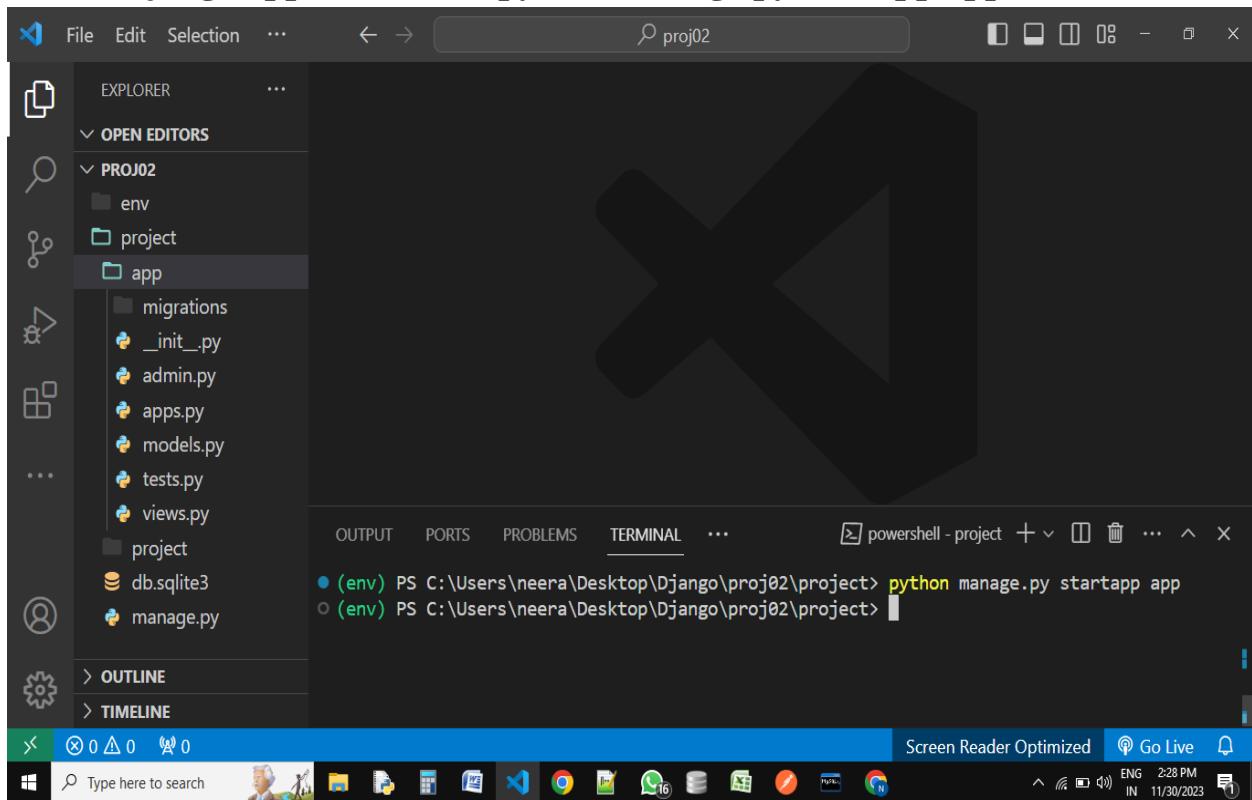
Project in Django

- A project in Django is a python package that represents the whole web application.
- A project in Django basically contains the configuration and setting related to the entire website.
- A single project can also have multiple apps in it that can be used to implement some functionality.

App in Django

- An app in Django is a sub-module of a project, and it is used to implement some functionality.
- Now, you can refer to an app as a standalone python module that is used to provide some functionality to your project.
- We can create multiple apps within a single Django project. And these apps can be independent of one another. Theoretically, we can use an app from one Django project to another without making any changes to it.

Create Django applications:-- (python manage.py startapp app)



Folder structure of Django-App

```
app/
|   __init__.py
|   admin.py
|   apps.py
|   models.py
|   views.py
|   test.py
```

1. __init__.py

This file provides the same functionality as that in the __init__.py file in the Django project structure. It is an empty file and does not need any modifications. It just represents that the app directory is a package.

2. admin.py

Admin.py file is used for registering the Django models into the Django administration. It is used to display the Django model in the Django admin panel. It performs three major tasks:

- a. Registering models
- b. Creating a Superuser (python manage.py createsuperuser ----enter)
- c. Logging in and using the web application

3. apps.py

Apps.py is a file that is used to help the user include the application configuration for their app. Users can configure the attributes of their application using the apps.py file. However, configuring the attributes is a rare task a user ever performs, because most of the time the default configuration is sufficient enough to work with.

4. models.py

Models.py represents the models of web applications in the form of classes. It is considered the most important aspect of the App file structure. Models define the structure of the database. It tells about the actual design, relationships between the data sets, and their attribute constraints.

5. views.py

Views are also an important part when we talk about the Django app structure. Views provide an interface through which a user interacts with a Django web application. It contains all the views in the form of classes. We use the concept of Serializers in Django Rest_Framework for making different types of views. Some of these are CustomFilter Views, Class-Based List Views, and Detail Views.

6. tests.py

Tests.py allows the user to write test code for their web applications. It is used to test the working of the app. Its working is quite complex. We will discuss it in more detail in the upcoming articles.

Installed app:-----

1. django.contrib.admin

- **Purpose:** It gives you a ready-made, web-based dashboard to manage your website's data easily.
- **Functionality:** With this dashboard, you can create, view, update, and delete data (like blog posts, user profiles, etc.) through a simple interface. It also lets you customize how things look and work in the admin panel.
- **Usage:** This tool is mainly used by the people who run the website to manage everything, like content, users, and settings, without needing to write any code.

2. django.contrib.auth

- **Purpose:** It helps your website handle user login, registration, and permissions.
- **Functionality:** It provides the tools needed to manage users, their passwords, who can access what (permissions), and user groups. This includes everything from signing up and logging in to resetting passwords and controlling who can see or do certain things on the site.
- **Usage:** It's used to keep your website secure by controlling who can access different parts and features, making sure only the right people can do certain actions.

3. django.contrib.contenttypes

Purpose: Enables generic relationships between models.

- **Purpose:** It lets different parts of your website connect with each other in a flexible way.
- **Functionality:** It provides tools that allow one type of data (like a comment) to link to any other type of data (like a blog post, product, or user) without being tied to just one specific type. This is useful for things like tagging, comments, or activity feeds that can be connected to various kinds of content.

- **Usage:** You use it when you need to create connections between different types of data on your site, making it easy to relate different models to each other without restrictions.

4. django.contrib.sessions

- **Purpose:** Manages user sessions, which track the state of a user across multiple requests.
- **Functionality:** Stores session data on the server-side, typically in the database, using a session key that is stored in a user's cookie. Sessions allow you to persist information (like user login status, shopping cart contents, etc.) across different requests from the same user.
- **Usage:** Used to maintain state between requests, which is essential for implementing features like user login, shopping carts, and other personalized experiences.

5. django.contrib.messages

- **Purpose:** Provides a framework for sending temporary, one-time messages to the user.
- **Functionality:** Integrates with the request/response cycle to display messages (such as success, warning, or error messages) to users after certain actions, like form submissions. Messages are typically displayed on the next page the user visits.
- **Usage:** Used to inform users of the results of their actions, such as confirming successful form submissions, warning about errors, or providing informational messages.

6. django.contrib.staticfiles

- **Purpose:** Manages the serving and collection of static files (CSS, JavaScript, images) for your application.
- **Functionality:** Handles the storage and retrieval of static files during development and production. It provides commands for collecting static files from various apps and locations into a single directory for easier deployment.
- **Usage:** Used to manage and serve static content required by your application, ensuring that these files are correctly handled across different environments (development, staging, production).

Summary of Differences:

- **django.contrib.admin:** Focuses on providing a graphical interface for managing your application's data and models.
- **django.contrib.auth:** Manages user authentication and authorization, including login, registration, and permissions.
- **django.contrib.contenttypes:** Enables dynamic, model-agnostic relationships using generic foreign keys.
- **django.contrib.sessions:** Manages session data, allowing for user state persistence across requests.
- **django.contrib.messages:** Provides a system for displaying one-time messages to users, often after form submissions or other actions.
- **django.contrib.staticfiles:** Handles static file management, including collecting, storing, and serving CSS, JavaScript, and images.

Each of these components plays a unique role in building a robust and functional Django application, covering a wide range of common web development needs.

In Django, responses are typically sent back to the client from views. Here are some common types of responses you can generate in Django:

1. HttpResponse

- The most basic type of response, sending plain text or HTML back to the client.

Example:

```
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect("Hello, world!")
```

2. JsonResponse

- A subclass of HttpResponse that automatically serializes Python dictionaries to JSON.

Example:

```
from django.http import JsonResponse

def my_view(request):
    data = {'key': 'value'}
    return JsonResponse(data)
```

3. HttpResponseRedirect

- A response that redirects the client to a different URL.

Example:

```
from django.http import HttpResponseRedirect
from django.urls import reverse
def my_view(request):
    return HttpResponseRedirect(reverse('login'))
    return HttpResponseRedirect('/another-url/')
```

4. HttpResponsePermanentRedirect

- Similar to HttpResponseRedirect, but sends a permanent redirect status (301).

Example:

```
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect('/some/url/')
```

5. HttpResponseNotFound

- A response for returning a 404 status code, indicating that a resource was not found.

Example:

```
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect("Page not found")
```

6. HttpResponseForbidden

- A response for returning a 403 status code, indicating that access to the resource is forbidden.

Example:

```
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect("Forbidden access")
```

7. HttpResponseServerError

- A response for returning a 500 status code, indicating a server error.

Example:

```
from django.http import HttpResponseServerError

def my_view(request):
    return HttpResponseServerError("Server error")
```

8. FileResponse

- A response used to send a file to the client, typically for downloading.

Example:

```
from django.http import FileResponse

def download_file(request):
    file_path = '/path/to/your/file.pdf'
    return FileResponse(open(file_path, 'rb'))
```

9. StreamingHttpResponse

- A response that streams data to the client, often used for large files or real-time data.

Example:

```
from django.http import StreamingHttpResponse

def stream_response(request):
    def stream_generator():
        for i in range(100):
            yield f"Line {i}\n"
    return StreamingHttpResponse(stream_generator())
```

10. TemplateResponse

- A response that renders a template and returns the rendered HTML. It can be useful when you want to delay template rendering.

Example:

```
from django.template.response import TemplateResponse

def my_view(request):
    context = {'key': 'value'}
    return TemplateResponse(request, 'my_template.html', context)
```

11. HttpResponseRedirect

- A response for returning a 302 status code, indicating a redirect.

Example:

```
python
Copy code
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect("http://example.com")
```

12. HttpResponseNotAllowed

- A response for returning a 405 status code, indicating that the method is not allowed.

Example:

```
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect("http://example.com")
```

Static, Template, and Media in Django

Static and Template reside:-

Out-side app:-

Project/settings.py:-

for templates

```
Temp_path = os.path.join(BASE_DIRS,'templates')
```

```
# add this variable to templates section
```

```
TEMPLATES = [
```

```
    'DIRS':[ Temp_path],
```

```
]
```

For static

```
Static_path = os.path.join(BASE_DIRS,'static')
```

```
STATICFILES_DIRS = ['Static_path'],
```

MEDIA_ROOT:

This setting defines the absolute file system path where Django will store uploaded files. It's the actual location on your server where the files will reside.

MEDIA_URL:

This setting defines the public URL that users will use to access the uploaded files. It's how you reference these files in your templates or code.

MEDIA_URL = 'media/'

MEDIA_ROOT = os.path.join(BASE_DIRS,'media')

Project/urls.py:-

```
from django.conf import settings
```

```
from django.conf.urls.static import static
```

```
urlpatterns = [
```

```
] + static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

Access image

```
<img src='media/{{----}}' style=' ' alt=' ' />
```

File access

```
<a href="{{ ----.url }}'' download></a>
```

:-DTL Tags:-

Without closing tag

1. { % csrf_token % }
2. { % extends % }
3. { % include % }
4. { % load % }
5. { % url % }
6. { % cycle % }

With closing tag

7. { % with % }
 { % endwith % }
8. { % if (condition) % }
 { % elif (condition) % }
 { % else % }
 { % endif % }
9. { % for x in condition % }
 { % endfor % }
10. { % block content % }
 { % endblock % }
11. { % comment % }
 { % endcomment % }

12. for ... empty

The for tag can take an optional { % empty % } clause whose text is displayed if the given array is empty or could not be found:

```
<ul>  
  { % for athlete in athlete_list % }  
    <li>{ { athlete.name } }</li>  
  { % empty % }  
    <li>Sorry, no athletes in this list.</li>  
  { % endfor % }  
</ul>
```

The above is equivalent to – but shorter, cleaner, and possibly faster than – the following:

```
<ul>  
  { % if athlete_list % }  
    { % for athlete in athlete_list % }  
      <li>{ { athlete.name } }</li>  
    { % endfor % }  
  { % else % }  
    <li>Sorry, no athletes in this list.</li>  
  { % endif % }  
</ul>
```

Built-in filter

Add:---

Adds the argument to the value.

For example:

```
{ { value|add:"2" } }
```

If **value** is **4**, then the output will be **6**.

This filter will first try to coerce both values to integers. If this fails, it'll attempt to add the values together anyway. This will work on some data types (strings, list, etc.) and fail on others. If it fails, the result will be an empty string.

For example, if we have:

```
{ { first|add:second } }
```

and **first** is **[1, 2, 3]** and **second** is **[4, 5, 6]**, then the output will be **[1, 2, 3, 4, 5, 6]**.

lower

Converts a string into all lowercase.

For example:

```
{ { value|lower } }
```

If value is **Totally LOVING this Album!**, the output will be **totally loving this album!**.

upper

Converts a string into all uppercase.

For example:

```
{ { value|upper } }
```

If value is **"Joel is a slug"**, the output will be **"JOEL IS A SLUG"**.

Capfirst:----

Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

For example:

```
{% value|capfirst %}
```

If **value** is "django", the output will be "Django".

Center:----

Centers the value in a field of a given width.

For example:

```
<pre>"{{ value|center:"15" }}"</pre>
```

If **value** is "Django", the output will be " Django ".

ljust:-

Left-aligns the value in a field of a given width.

Argument: field size

For example:

```
<pre>"{{ value|ljust:"10" }}"</pre>
```

If value is Django, the output will be "Django ".

rjust:-

Right-aligns the value in a field of a given width.

Argument: field size

For example:

```
<pre>"{{ value|rjust:"10" }}"</pre>
```

If value is Django, the output will be " Django".

cut:----

Removes all values of arg from the given string.

For example:

```
{ { value|cut:" " } }
```

If **value** is "**String with spaces**", the output will be "**Stringwithspaces**".

last:-

Returns the last item in a list.

For example:

```
{ { value|last } }
```

If value is the list ['a', 'b', 'c', 'd'], the output will be the string "d".

length:---

Returns the length of the value. This works for both strings and lists.

For example:

```
{ { value|length } }
```

If value is ['a', 'b', 'c', 'd'] or "abcd", the output will be 4.

The filter returns 0 for an undefined variable.

linenumbers:---

Displays text with line numbers.

For example:

```
{ { value|linenumbers } }
```

If value is:

```
one  
two  
three
```

the output will be:

1. one
2. two
3. three

join

Joins a list with a string, like Python's str.join(list)

For example:

```
{ { value|join:" // " } }
```

If value is the list ['a', 'b', 'c'], the output will be the string "a // b // c".

random

Returns a random item from the given list.

For example:

```
{ { value|random } }
```

If value is the list ['a', 'b', 'c', 'd'], the output could be "b".

slice

Returns a slice of the list.

Uses the same syntax as Python's list slicing. See the [Python documentation](#) for an introduction.

Example:

```
{ { some_list|slice:"2" } }
```

If some_list is ['a', 'b', 'c'], the output will be ['a', 'b'].

title

Converts a string into titlecase by making words start with an uppercase character and the remaining characters lowercase. This tag makes no effort to keep “trivial words” in lowercase.

For example:

```
{ { value|title } }
```

If value is "my FIRST post", the output will be "My First Post".

truncatechars

Truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis character ("...").

Argument: Number of characters to truncate to

For example:

```
{ { value|truncatechars:7 } }
```

If value is "Joel is a slug", the output will be "Joel i...".

truncatewords

Truncates a string after a certain number of words.

Argument: Number of words to truncate after

For example:

```
{ { value|truncatewords:2 } }
```

If value is "Joel is a slug", the output will be "Joel is ...".

Newlines within the string will be removed.

wordcount

Returns the number of words.

For example:

```
{ { value|wordcount } }
```

If value is "Joel is a slug", the output will be 4.

The for loop sets a number of variables available within the loop:

Variable	Description
forloop.counter	The current iteration of the loop (1-indexed)
forloop.counter0	The current iteration of the loop (0-indexed)
forloop.revcounter	The number of iterations from the end of the loop (1-indexed)
forloop.revcounter0	The number of iterations from the end of the loop (0-indexed)
forloop.first	True if this is the first time through the loop
forloop.last	True if this is the last time through the loop
forloop.parentloop	For nested loops, this is the loop surrounding the current one

default

If value evaluates to **False**, uses the given default. Otherwise, uses the value.

For example:

```
{ { value|default:"nothing" } }
```

If **value** is "" (the empty string), the output will be **nothing**.

default if none

If (and only if) value is **None**, uses the given default. Otherwise, uses the value.

Note that if an empty string is given, the default value will *not* be used. Use the **default** filter if you want to fallback for empty strings.

For example:

```
{ { value|default_if_none:"nothing" } }
```

If **value** is **None**, the output will be **nothing**.

dictsort

Takes a list of dictionaries and returns that list sorted by the key given in the argument.

For example:

```
{ { value|dictsort:"name" } }
```

If value is:

```
[  
  {"name": "zed", "age": 19},  
  {"name": "amy", "age": 22},  
  {"name": "joe", "age": 31},  
]
```

then the output would be:

```
[  
  {"name": "amy", "age": 22},  
  {"name": "joe", "age": 31},  
  {"name": "zed", "age": 19},  
]
```

divisibleby

Returns True if the value is divisible by the argument.

For example:

```
{ { value|divisibleby:"3" } }
```

If value is 21, the output would be True.

floatformat

When used without an argument, rounds a floating-point number to one decimal place – but only if there's a decimal part to be displayed. For example:

value	Template	Output
34.23234	<code>{{ value floatformat }}</code>	34.2
34.00000	<code>{{ value floatformat }}</code>	34
34.26000	<code>{{ value floatformat }}</code>	34.3

If used with a numeric integer argument, **floatformat** rounds a number to that many decimal places. For example:

value	Template	Output
34.23234	<code>{{ value floatformat:3 }}</code>	34.232
34.00000	<code>{{ value floatformat:3 }}</code>	34.000
34.26000	<code>{{ value floatformat:3 }}</code>	34.260

Particularly useful is passing 0 (zero) as the argument which will round the float to the nearest integer.

value	Template	Output
34.23234	<code>{{ value floatformat:"0" }}</code>	34
34.00000	<code>{{ value floatformat:"0" }}</code>	34
39.56000	<code>{{ value floatformat:"0" }}</code>	40

If the argument passed to **floatformat** is negative, it will round a number to that many decimal places – but only if there's a decimal part to be displayed. For example:

value	Template	Output
34.23234	<code>{{ value floatformat:"-3" }}</code>	34.232
34.00000	<code>{{ value floatformat:"-3" }}</code>	34
34.26000	<code>{{ value floatformat:"-3" }}</code>	34.260

If the argument passed to **floatformat** has the **g** suffix, it will force grouping by the **THOUSAND_SEPARATOR** for the active locale. For example, when the active locale is **en** (English):

value	Template	Output
34232.34	<code>{{ value floatformat:"2g" }}</code>	34,232.34
34232.06	<code>{{ value floatformat:"g" }}</code>	34,232.1
34232.00	<code>{{ value floatformat:"-3g" }}</code>	34,232

Output is always localized (independently of the **{% localize off %}** tag) unless the argument passed to **floatformat** has the **u** suffix, which will force disabling localization. For example, when the active locale is **pl** (Polish):

value	Template	Output
34.23234	<code>{{ value floatformat:"3" }}</code>	34,232
34.23234	<code>{{ value floatformat:"3u" }}</code>	34.232

Using **floatformat** with no argument is equivalent to using **floatformat** with an argument of **-1**.

slugify

Converts to ASCII. Converts spaces to hyphens. Removes characters that aren't alphanumerics, underscores, or hyphens. Converts to lowercase. Also strips leading and trailing whitespace.

For example:

```
{ { value|slugify } }
```

If value is "Joel is a slug", the output will be "joel-is-a-slug".

Type of views in DRF:---

1. Function Based Views

2. Class-based Views

1. Function Based Views:---

REST framework also allows you to work with regular function based views. It provides a set of simple decorators that wrap your function based views to ensure they receive an instance of **Request** (rather than the usual Django **HttpRequest**) and allows them to return a **Response** (instead of a Django **HttpResponse**), and allow you to configure how the request is processed.

```
@api_view()
```

Signature: @api_view(http_method_names=['GET'])

```
from rest_framework.decorators import api_view
```

```
from rest_framework.response import Response
```

```
@api_view() # default method is 'GET', that means no need to written here.
```

```
def hello_world(request):
```

```
    return Response({"message": "Hello, world!"})
```

```
@api_view(['GET', 'POST'])
```

```
def hello_world(request):
```

```
    if request.method == 'POST':
```

```
        return Response({"message": "Got some data!", "data": request.data})
```

```
    return Response({"message": "Hello, world!"})
```

2. Class-based Views :----

REST framework provides an APIView class, which subclasses Django's View class. APIView classes are different from regular View classes in the following ways:

1. Requests passed to the handler methods will be REST framework's Request instances, not Django's HttpRequest instances.
2. Handler methods may return REST framework's Response, instead of Django's HttpResponse.
3. The view will manage content negotiation and setting the correct renderer on the response.
4. Any API Exception exceptions will be caught and mediated into appropriate responses.
5. Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.
6. Using the APIView class is pretty much the same as using a regular View class, as usual, the incoming request is dispatched to an appropriate handler method such as .get() or .post().
7. Additionally, a number of attributes may be set on the class that controls various aspects of the API policy.