**Class 1 Topic : Django Basics (Installations, Requirements,)**

**Class 2 Topic : - Virtual Environment,Requirement.txt, Django-Admin Panel , Django apps (Purpose of Creating Django Apps), Django Models, Diff btw Django Project and Django Apps, Introduction To Django ORM ,MVT-MVC Framework model …**

**1. Virtual Environment :-** Python, like most other modern programming languages, has its own unique way of downloading, storing, and resolving packages (or [modules](https://en.wikipedia.org/wiki/Modular_programming)). These packages are installed on your system. **However** third party packages installed by using [easy\_install](https://pythonhosted.org/setuptools/easy_install.html) or [pip](https://en.wikipedia.org/wiki/Pip_(package_manager)) are typically placed in one of the directories pointed to by [site.getsitepackages](https://docs.python.org/3/library/site.html#site.getsitepackages):

\* It’s important to know this because, **by default, every project on your system will use these same directories to store and retrieve site packages (3rd party libraries).**

**\* Consider the following scenario where you have two projects - ProjectA and ProjectB, both of which have a dependency on the same library, ProjectC. The problem becomes apparent when we start requiring different versions of ProjectC. Maybe ProjectA needs v1.0.0, while ProjectB requires the newer v2.0.0,** for example.

\* This is a real problem for **Python since it can’t differentiate between versions in the “site-packages” directory. So both v1.0.0 and v2.0.0 would reside in the same directory with the same name:** And since projects are stored according to just their name there is no differentiation between versions. Thus, both projects**, ProjectA and ProjectB, would be required to use the same version, which is unacceptable in many cases.**

\* This is where the concept of virtual environments (and the [virtualenv](https://virtualenv.readthedocs.org/en/latest/)/[venv](https://docs.python.org/3/library/venv.html) tools) comes into play…

**1.1 What is a virtual environment?**

At its core, the main purpose of Python virtual environments is **to create an isolated environment for Python projects.** This means that each project can have its own dependencies, regardless of what dependencies every other project has.

**2. Requirement.txt -- 🡪**

pip freeze > requirements.txt without activating virtual environment and with virtual environment

pip install -r requirements.txt

python manage.py runserver

You have unapplied migrations; your app may not work properly until they are applied. Run 'python manage.py migrate' to apply them

**2. Django - Admin Panel :-**

hit the url := http://127.0.0.1:8000/admin/

To log in, you need to create a superuser - a user account that has control over everything on the site. Go back to the command line, type python manage.py createsuperuser, and press enter.

python manage.py createsuperuser

**operational error :- no such table: django\_session**

**python manage.py migrate**

**python manage.py makemigrations**

# Migrations

**Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema.** They’re designed to be mostly automatic, but you’ll need to know when to make migrations, when to run them, and the common problems you might run into.

**The Commands**

**makemigrations** basically generates the SQL commands for preinstalled apps (which can be viewed in installed apps in settings.py) and your newly created apps' model which you add in installed apps.**It does not execute those commands in your database file.** So tables doesn't created after makemigrations.

**migrate** executes those SQL commands in database file. So after executing migrate all the tables of your installed apps are created in your database file.

* [**migrate**](https://docs.djangoproject.com/en/2.0/ref/django-admin/#django-admin-migrate), which is responsible for applying and unapplying migrations.
* [**makemigrations**](https://docs.djangoproject.com/en/2.0/ref/django-admin/#django-admin-makemigrations), which is responsible for creating new migrations based on the changes you have made to your models.
* [**sqlmigrate**](https://docs.djangoproject.com/en/2.0/ref/django-admin/#django-admin-sqlmigrate), which displays the SQL statements for a migration.
* [**showmigrations**](https://docs.djangoproject.com/en/2.0/ref/django-admin/#django-admin-showmigrations), which lists a project’s migrations and their status.

**The migration files for each app live in a “migrations” directory inside of that app, and are designed to be committed to, and distributed as part of, its codebase**. You should be making them once on your development machine and then running the same migrations on your colleagues’ machines, your staging machines, and eventually your production machines.

 Creating a table

 Adding/Removing a column from the table

 Updating table/column name

 Removing a table/column

### 3. Creating an Django application

To keep everything tidy, we will create a separate application inside our project. It is very nice to have everything organized from the very beginning. To create an application we need to run the following command in the console (from Your project directory where manage.py file is):

python manage.py startapp Your\_App\_Name

After creating an application, we also need to tell Django that it should use it. We do that in the file My\_Project/settings.py. We need to find INSTALLED\_APPS and add a line containing 'Your\_App\_Name', just above ]. So the final product should look like this:

My\_Project/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

' Your\_App\_Name',

]

# 4. Django - Models

A model is a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection. Models are defined in the mukeshapp/models.py (in our example: mukeshapp/models.py)

As we know Django is an [ORM](https://en.wikipedia.org/wiki/Object-relational_mapping) (Object Relational Mapping). When we use the command:

python manage.py makemigrations [app\_name]

It will generate the sql command to create the table corresponding to each class you made in models.py file. then the command:

python manage.py migrate [app\_name]

will create the table in database using the commands which have been generated by makemigrations.

For example, if we make a model class-

from django.db import models

class Person(models.Model):

first\_name = models.CharField(max\_length=30)

last\_name = models.CharField(max\_length=30)

The corresponding sql command after using makemigrations will be

CREATE TABLE myapp\_person (

"id" serial NOT NULL PRIMARY KEY,

"first\_name" varchar(30) NOT NULL,

"last\_name" varchar(30) NOT NULL

);

and using above command, table will be created in the database when we use migrate.

## Creating a Model

from django.db import models

*# Create your models here.***class** UserModel(models.Model): # We need to inherit the parent class for our custom model to be a valid model which is models.Model.  
 email = models.EmailField()  
 name = models.CharField(max\_length=120)  
 username = models.CharField(max\_length=120)  
 password = models.CharField(max\_length=40)  
 created\_on = models.DateTimeField(auto\_now\_add=**True**)  
 updated\_on = models.DateTimeField(auto\_now=**True**)

def \_\_str\_\_(self):

return self.email + " " + self.username

Double-check that you use two underscore characters (\_) on each side of str. This convention is used frequently in Python and sometimes we also call them "dunder" (short for "double-underscore

**\_\_**str\_\_() The **\_\_**str\_\_() method is called whenever you call **str**() on an object.**Django** uses **str**(obj) in a number of places. Most notably, to display an object in the**Django** admin site and as the value inserted into a template when it displays an object.

*# Create your models here.  
# class User(models.Model):  
# name = models.CharField(max\_length=255)  
# phone = models.CharField(max\_length=30)  
# age = models.IntegerField(default=0) # Default in case user doesn’t provide any .  
# has\_verified\_mobile = models.BooleanField(default=False)  
# created\_on = models.DateTimeField(auto\_now\_add=True)*

Setting auto\_now\_add attribute to True tells django whether it should automatically add a date of the current time when a new user is created or it

python manage.py makemigrations yourapp

python manage.py migrate

# 5. Django admin

To add, edit and delete the posts we've just modeled, we will use Django admin.

Let's open the yourapp/admin.py file and replace its contents with this:

YourApp/admin.py

from django.contrib import admin

from .models import your\_model\_name

admin.site.register(register your model name )

at this end we already created our superuser .so just login and check your registered model.

(https://tutorial.djangogirls.org/en/deploy/)

# Django ORM and QuerySets

One of the most powerful features of Django is its Object-Relational Mapper (ORM), which enables you to interact with your database, like you would with SQL. In fact, Django's ORM is just a pythonical way to create SQL to query and manipulate your database and get results in a pythonic fashion.

Before we start looking into how the ORM works, we need a database to manipulate. As with any relational database, we need to define a bunch of tables and their relationships (i.e., the way they relate to each other). Let's use something familiar. For example, say we want to model a user that has name,age,username,phonenumber.

## What is a QuerySet?

A QuerySet is, in essence, a list of objects of a given Model. QuerySets allow you to read the data from the database, filter it and order it.

## Django shell

Open up your terminal and type this command:(Don’t forget to activate virtualenv if you are in)

python manage.py shell

The effect should be like this:

command-line

(InteractiveConsole)

>>>

You're now in Django's interactive console. It's just like the Python prompt, but with some additional Django magic. :) You can use all the Python commands here too, of course.

### All objects

Let's try to display all of our user first. You can do that with the following command:

command-line

>>> UserModel.objects.all()

Traceback (most recent call last):

File "<console>", line 1, in <module>

NameError: name 'user' is not defined

Oops! An error showed up. It tells us that there is no user. It's correct – we forgot to import it first!

command-line

>>> from yourapp.models import UserModel

We import the model user from yourapp.models. Let's try displaying all posts again:

command-line

>>> UserModel.objects.all()

<QuerySet [ ]>

This is a list of the users we created earlier! We created these posts using the Django admin interface. But now we want to create new posts using Python, so how do we do that?

### Create object

This is how you create a new user object in database:

command-line

>>> new\_user = User(name="Mukesh", age=23, has\_verified\_mobile= True, phone="1234567890", )

new\_user = UserModel(name="Mukesh", email="Mukeshdubey@gmail.com", username= "im\_mukeshdubey", password="123456789", )

Next, we need to actually CREATE or save the data to the database:  
new\_user.save()

On calling save() the django ORM generates the following SQL Equivalent query and runs it on the db:  
   
INSERT INTO UserModel (name, age, has\_verified\_mobile, phone) VALUES ('Mukesh', 23, True, '1234567890') ;

lets chk it actually added or not ..

Let's run our first query to READ data from the table (Read):  
existing\_user = UserModel.objects.get(id=1)

On calling get() the django ORM generates the following SQL Equivalent query, runs it on the db and returns the result:  
Select \* from user where id = 1 ;

print (existing\_user.name)  
 print (existing\_user.username)

(https://tutorial.djangogirls.org/en/django\_orm/)

use exit() command to exit from interactive console

Diff Btw Project and Application…>

Technically you could create a project with no apps and just write all of your code in the "project" folder, you would be missing out on a lot of the power of Django. The Django project stresses the DRY principle (meaning Don't Repeat Yourself), and the idea of apps is a big portion of that.

your project should be composed by apps. That's why when using the command line, you create a project, an later on, add apps to that project.  you may use django to write many websites. It would make sense that each new website is a new project. But, all of your websites have some things in common. They all have comment sections, and they all have user registration. So you could write a comment app that you would then install in each of the projects. You may also want to write an app for registering users. It's not necessary to create apps for every action on your site, but there are some common patterns that arise.

A *project* refers to the entire application and all its parts.

An *app* refers to a submodule of the project. It's self-sufficient and not intertwined with the other apps in the project

An app typically has it's own models.py (which might actually be empty). You might think of it as a standalone python module. A simple project might only have one app.

For your example, the project is the whole website. You might structure it so there is an app for music sections, and an app for videos, and an app for settings or anything. If they need to interact with each other, they do it through well-documented public classes and accesser methods.

 Django is a Python web framework. And like most modern framework, Django supports the MVC pattern. First let's see what is the Model-View-Controller (MVC) pattern, and then we will look at Django’s specificity for the Model-View-Template (MVT) pattern.

MVC Pattern

When talking about applications that provides UI (web or desktop), we usually talk about MVC architecture. And as the name suggests, MVC pattern is based on three components: Model, View, and Controller.

DJANGO MVC - MVT Pattern

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that **Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template.** The template is a HTML file mixed with Django Template Language (DTL).

The following diagram illustrates how each of the components of the MVT pattern interacts with each other to serve a user request −



The developer provides the Model, the view and the template then just maps it to a URL and Django does the magic to serve it to the user.

Django follows MVC pattern very closely but it uses slightly different terminology. Django is essentially an MTV (Model-Template-View) framework. Django uses the term Templates for Views and Views for Controller. In other words, in Django views are called templates and controllers are called views. Hence our HTML code will be in templates and Python code will be in views and models.

https://overiq.com/django/1.10/mvc-pattern-and-django/

# Class 3 (Django Bacics)

* HTML Forms
* Static files
* Django Forms
* User Login

**HTML Forms :**

An HTML form contains form elements.

Form elements are different types of input elements, like text fields, checkboxes, radio buttons, submit buttons, and more.

1 Create a template (form.html).

<!DOCTYPE html>

<html lang="en">

<head>

 <title>{{ title }}</title>

 <meta charset="utf-8">

 <meta name="viewport" content="width=device-width, initial-scale=1">

 <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

 <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

</head>

<body>

<div class="container">

 <h2>{{ content}}</h2>

<form method='POST'> {% csrf\_token %}

   <div class="form-group">

     <label for="email">Email:</label>

     <input type="email" class="form-control" id="email" placeholder="Enter email" name="email">

   </div>

   <div class="form-group">

     <label for="pwd">Password:</label>

     <input type="password" class="form-control" id="pwd" placeholder="Enter password" name="pwd">

   </div>

   <div class="checkbox">

     <label><input type="checkbox" name="remember"> Remember me</label>

   </div>

   <button type="submit" class="btn btn-default">Submit</button>

 </form>

</div>

</body>

</html>

CSRF Token :- Cross-Site Request Forgery (CSRF)

Yes, the post data is safe. But the origin of that data is not. This way somebody can trick user with JS into logging in to your site, while browsing attacker's web page.

In order to prevent that, django will send a random key both in cookie, and form data. Then, when users POSTs, it will check if two keys are identical. In case where user is tricked, 3rd party website cannot get your site's cookies, thus causing auth error.

Visit :- https://stackoverflow.com/questions/5207160/what-is-a-csrf-token-what-is-its-importance-and-how-does-it-work

2 Create a function in **views.py**for form connectivity with django .

from django.contrib.auth import authenticate , login

from django.http import HttpResponse       # Returns response

from django.shortcuts import render , redirect

from .forms import contactForm ,LoginForm

# html form

def firstform(request):

   context={

   "title": "contact",

   "content":"welcome to contact page",

   }

    if request.method == "POST" :

       print(request.POST)

       print(request.POST.get('fullname'))

       print(request.POST.get('email'))

       print(request.POST.get('content'))

    return render(request, "form.html" , context)

3 Create a Url for template form.html in **urls.py**

from django.contrib import admin

from django.urls import path

from django.conf.urls import url

from .views import firstform ,login\_page

urlpatterns = [

   url(r'^admin/',admin.site.urls),

   url(r'^forms',firstform),

   url(r'^login',login\_page),

# Managing static files (e.g. images, JavaScript, CSS)

In web application, apart from business logic and data handling, we also need to handle and manage static resources like: CSS, JavaScript, images etc in our application. In Django, we refer to these files as “static files”.

It is important to manage these resources so that it does not effect our application performance.

Django deals this very efficiently and provides a convenient manner to use resources.

Django provides [**django.contrib.staticfiles**](https://docs.djangoproject.com/en/2.0/ref/contrib/staticfiles/#module-django.contrib.staticfiles) to help you manage them.

## \* Configuring static files

## 

1. Include the **django.contrib.staticfiles** in **INSTALLED\_APPS.**

1.        INSTALLED\_APPS = [

2.            'django.contrib.admin',

3.            'django.contrib.auth',

4.            'django.contrib.contenttypes',

5.            'django.contrib.sessions',

6.            'django.contrib.messages',

7.            'django.contrib.staticfiles',

8.            'myapp'

9.        ]

2. Define STATIC\_URL in settings.py file. For example.

1.        STATIC\_URL = '/static/'

3. Load static files in templates by using the following code.

1.        {% load **static** %}

4. Store all images, JavaScript, CSS files in **static** folder of the application. First create a directory **static,** store the files inside it.

create a static folder inside application folder.

To, load static files in template file, use the following code. For example.

**// index.html**

1.          {% load **static** %}

2.        <!DOCTYPE html>

3.        <html lang="en">

4.        <head>

5.            <meta charset="UTF-8">

6.            <title>Index</title>

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

8.        </head>

9.        <body>

10.     <img src="{% static 'images/wallpaper.jpeg' %}" alt="My image" height="300px" width="700px"/>

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

11.     </body>

12.     </html>