**SECTION ONE (first\_test\_django\_project and first\_app)**

**Django Project**

* It act as a parent directory, it mean Overall an Web app or Web application
* To create the django project using django cli command : django-admin startproject <project\_name>
* This cmd create one folder(parent dir) with the givename, within that it will create one more folder(secondary directory) with the same name and manage.py file also created
* Few more files created under the second folder like \_\_init\_\_.py, setting.py,urls.py,asgi.py and wagi.py
* It is also called as **Project scope Files**
* To start the server : python manage.py runserver <PORT\_NUMBER> is used, it should run the parent directory

**Django App**

* It is sub-dir or sub-component
* It is main purpose is the make the huge project into multiple App/Module. For example, If we have Social Media application, we may have Message features, video call features. So we can put two app as message and video calling into single Django Project (Social Media Application)
* Goto the parent directory of the Django Project and create any number of apps into it.
* To create App : python manage.py startapp <APP\_NAME>
* This cmd will create a folder by given name and within that few files created like app.py admin.py , model.py , view.py and test.py
* It is also called as **App Scope Files**
* If needed or if our application big, we can create urls.py in the App Scope File
* We can have N no of App inside the Project
* Each app may have many views related files so we can configure it in the url.py which in the App Scoped Folders
* Then all Url.py file in the each app scoped folder, can be mapped into the url.py in the secondary folder which is created at the time of Project creation
* We can see the **first\_app** project which is done based the above two section.

**SECTION TWO (view\_route\_urls)**

**Views and URL**

* View means what information needs to show to the end-user
* URL means from where that information is shown to the end-user
* View is the webpage, using URL we can route to the desired view from the browser
* URL configuration can be in the both Project(Project scoped file) level and App(App scoped file) level using the urls.py file
* Using path() and include() Django functions we can configure the URLS in the urls.py files
* Urlpattern is the array type of variable used to wrap the path() and include() in the urls.py
* Connecting View to Url using path() function
* In the path() first argument is the **url** and second argument is the **view or function** and third optional arguments is **kwargs** and fourth optional arguments is **name**

**Dynamic Views**

* This is solution for the creating a each and every view whenever it is required
* Python Dictionary object will be helpful to make the View to render dynamically
* Passing the topic as an argument into the view we can render the dynamic view with the help of dictionary.
* Path convertor can be done in Django, it means, we can receive the any data type from the request

**Redirects and 404**

* In the dynamic view we have one issue, if any value given which is not in the dictionary, it gives error
* We should handle it by giving proper response.
* We can use the Django 404 page to that error
* Redirects used to redirect the one page to another page using Http

**URL names and reverse()**

* In the above redirects is not dynamic way so reserve() will be used
* For each url we can give a name
* That name can be used in the reverse() method across Django project

**Connecting to Templates**

* We wont want to type HTML code or HTML response directly into the view.py file
* To address this, template comes into the picture
* So we can move all out HTML files into separate folder and make the our view.py to communicate within them
* Recommended way to store the templates in the App level, but for now we can store it in the Project top level
* Edit the setting.py(TEMPLATES.DIRS key value) to inform Django Project to communicate the Template folder and View.py

**SECTION THREE(templates)**

**Template Directory**

* We should keep the template directory with the App Scoped Files
* It can be re-use in the multiple Django Projects wherever we need it
* To make aware the Template Directory in the App Scope to the Django Project , we should register the app in the INSTALLED\_APP keys in the setting.py
* STEPS :
  + Open cmd prompt and run “python manage.py migrate”
  + We need tell Django project , we have this app and needs to install, for that we should open the apps.py file in the corresponding each app in the project
  + In that file we can find a class has been generated and copy the name of the class
  + Goto setting.py file, add the path of this class in the INSTALLED\_APP variable
  + Run “python manage.py makemigrations <APP\_NAME>” we should run it for each and every app in our Django project.
  + Open cmd prompt and run “python manage.py migrate”
  + In setting.py make sure APP\_DIRS key needs to be true under the TEMPLATES Key
  + Create a folder under each app in the name of “templates” and also create a sub folder under this in the name of <CORRESPONDING\_APP\_NAME>
  + Place the html file in the created folder
  + Start the server and access the application

**Template Rendering / Django Template Language / Context Insertion:**

* We can pass the context value from the view to template.
* Context value can be any type, like variables, functions and another python file or re-direction logic and etc.
* In the render() we have one more parameter called “context”, it will pass the value to the html template.
* To access the passed value in the HTML, we should use syntax of double curly braces open and close {{ }}
* To access the python collection object the syntax should be little different , check in the official documentation

**Filter and Tags**

* Filters are built-in modifiers in Django templating that allow you to quickly apply a change to a variable on the template side, rather than in python script.
* We can place the filter in the template directly or same filter logic can be used in the view.py file also
* There are lot of easy and complex built-in filters available, please checkout in the official documentation.
* Tags are able to provide further logic at the template in rendering process.
* Tags includes for loop, if, else if and URL linking.
* Tags are denoted by {% %}

**URL Names in Templates**

* We should give one name to the app in the url.py file
* Give the name for each path
* Add <a>tag whereever we need and pass URL name in the href attributes like {% url ‘<app\_name>:<path\_name>’ %}

**Template Inheritance**

* Create a folder named “templates” at the top root level of the project
  + Or create folder named “templates” under the app level folder
* Create html file in this folder, which act as a base/parent html template
* Add {% block content %} {% endblock %}
* Create child html file inside the app level template folder
  + In this if you want to extend the base.html from the root level folder use {% extends '<HTML\_FILE\_NAME>' %}
  + In this if you want to extend the base.html from the app level folder use {% extends '<APP\_NAME>/<HTML\_FILE\_NAME>' %}

**Template Specifics – Static Files**

* Static file called as JS and CSS files
* Using Tags, Django can serve the static files instead of having full file path.
* To use the static file, in setting.py file double check the “Django.contrib.staticfiles” is available under the INSTALLED\_APPS
* And check the STATIC\_URL is available in the setting.py file, if you like to change the folder path, we can update under this name.
* As Template folder we should set the static folder
* Using {% load static%} and {% static <APP\_NAME>/<FILE\_NAME> %}

**Template Specific – Custom 404 file**

* Django comes up with the many build-in templates like 404 error page, admin page, user credentials entry page and etc
* But Django can able to overwrite these page based on our needed.
* We can do it the Project Level Template Directory and App Level Template Directory
* By default , it gives some unwanted information in the 404 error page.
* To Enable or show the Django 404 page or To show Django 404.html file under the Project level template folder, we should do the below changes in the settings.py
  + Change the Value of “DEBUG” from True to False
  + Add the IP address of localhost under the “ALOWED\_HOSTS”
* If no File is not under the name of 404.html, it will display the Django 404 html page.
* To use the custom error page other than 404.html filename, check the Django Documentation

**SECTION FOUR (models\_database)**

**Models and Database**

* It is used to interact with database with Pyhton(Django)
* We can do CURD operation by using models without native SQL quries
* Each model is equivalent One table
* Create a class and Insert this **“from django.db import models”**
* Each variable is considered as columns in the table
* Our Model file should inherit from the Python Model
* Using **fields** we can define both **data types** and **constraints**.
* We can able to create with Relationship between Two tables

**Creating Model and Fields**

* Create a new project and new one app.
* Register a database in the setting.py under the DATABASE keys,
* For this study we are going to use sqlite3 which is coming with python installation.
* Run Migrate cmd to create database using “python manage.py migrate”, it will generate db.sqlite3 file.
  + While running this cmd, Django will look into the INSTALLED\_APP in the setting.py
  + It will create tables in the Installed apps configure based on the Database engine configuration
  + If we configure our App under the INSTALLED\_APPS, it will check for models and create tables for us.
* Open the model.py file under the App scoped Files
* Create a class and inherit as mentioned above
* Create fields as needed

**Migration**

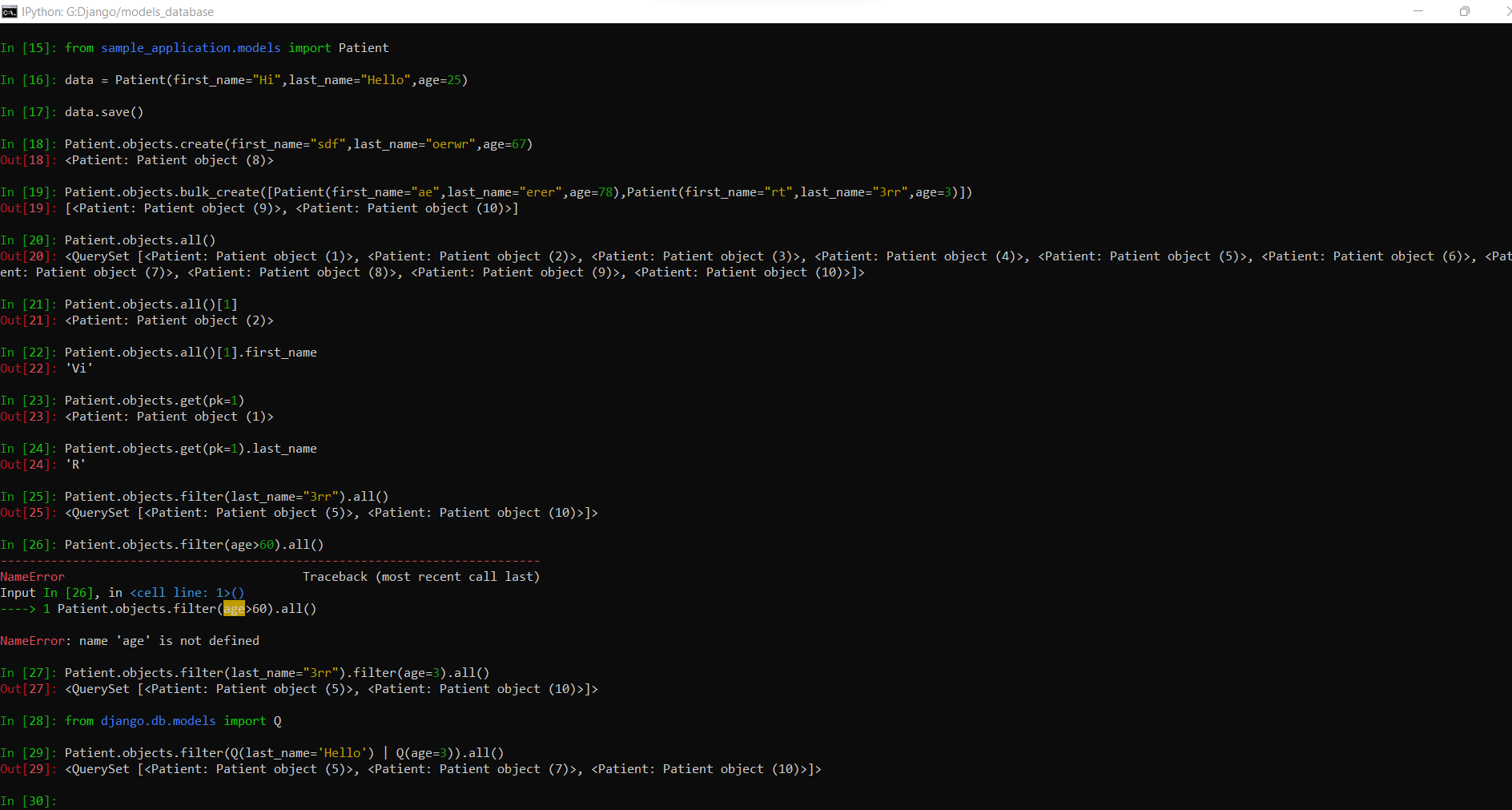
* Migration is used to connecting the Django Project/App to the Database.
* This will take of Add/Updating the models and it column/attributes
* Migration cmd will apply for the all app which is under the INSTALLED\_APP key in setting.py
* There are Three Migrate related commands
  + makemigrations
  + migrate
  + sqlmigrate
* **makemigrations**
  + CMD : python managed.py makemigrations <APP\_NAME>
  + This cmd will create set instruction python file to apply the changes to the database.
  + Default Application like Admin,Auth have its own makemigration code(Migration Instruction File) ready
  + This cmd does not make any change in the database
  + After running this cmd, the instruction file will be created under the <APP\_NAME>/migrations/<CREATED\_FILE\_NAME>
  + This file is human readable and if need we can also edit it
* **migrate**
  + CMD: python manage.py migration
  + This cmd will look into the migration folder which is created in the previous cmd
  + It will run all instruction by connecting the database and update the database
* **sqlmigrate**
  + CMD : python manage.py sqlmigration app <MIGRATION\_NUMBER>
  + TO view the file migration instruction in the SQL format we can run this command
  + This cmd wont do any change to the Database, it only show the migration instruction in the SQL queries format

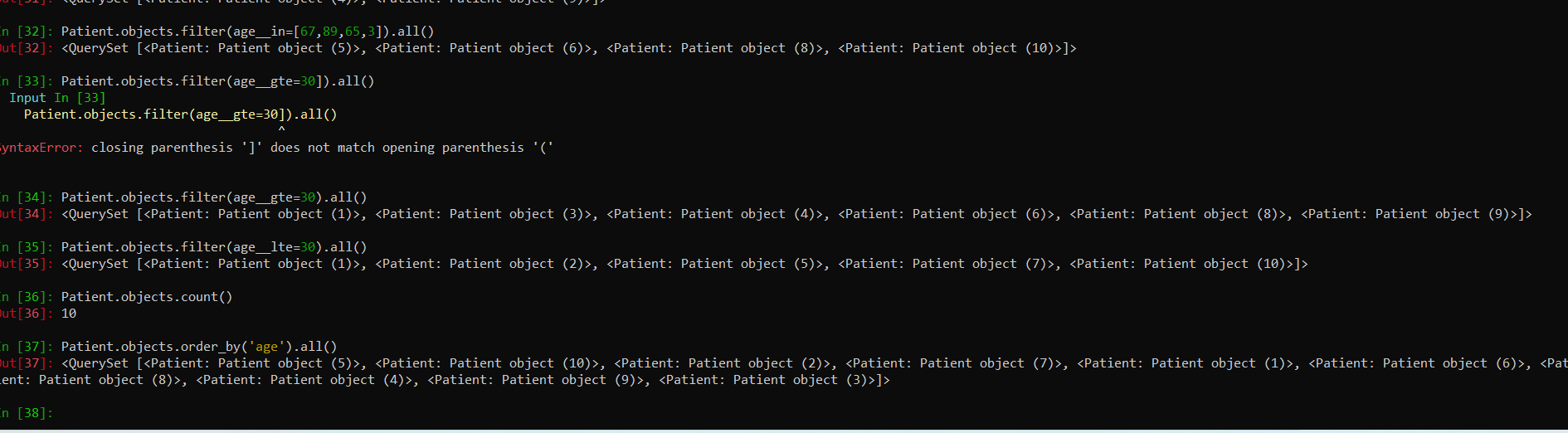
**Run Shell in Django**

* CMD : python manage.py shell

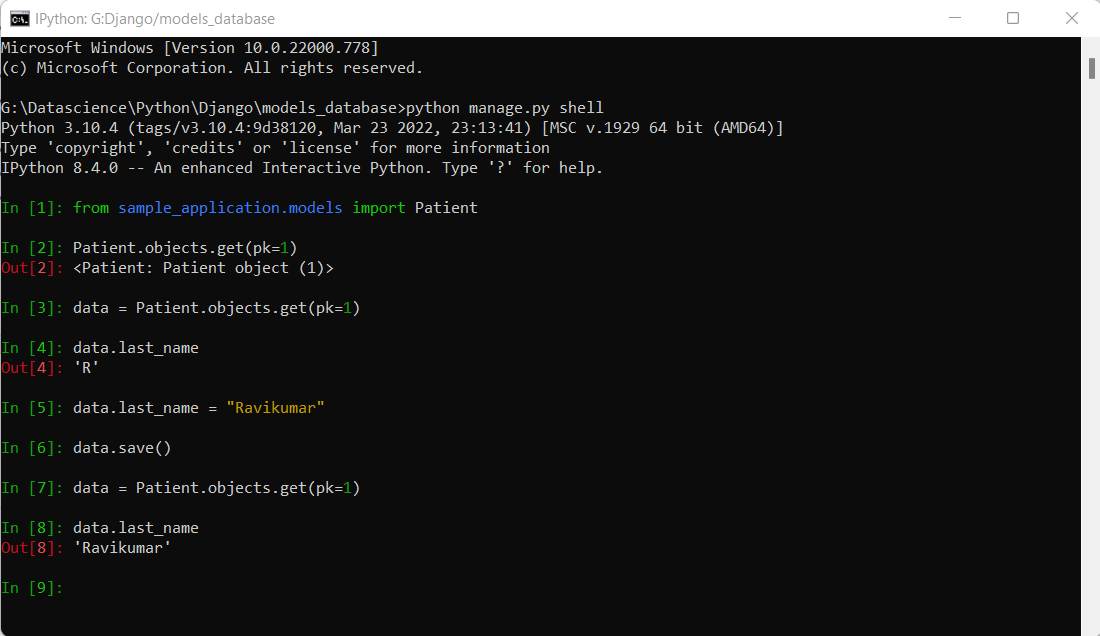
**Data Interaction (CURD Operation)**

* Save a record : <MODEL\_BOJECT>.save()
* Create and save record : <MODEL\_NAME>.objects.create()
* Save bulk record : <MODEL\_NAME>.objects.bulk\_create()
* NOTE : <MODEL\_NAME>.objects 🡺 This is called as Django Model Manager
* Using this we can also call more methods to do Database Operation using Django Models
* <MODEL\_NAME>.objects.all() and <MODEL\_NAME>.objects.get()
* <MODEL\_NAME>.objects.get() – It will grab only one item from the Table, by passing any unique key
* <MODEL\_NAME>.objects.filter() is used to narrow down the result if needed from the QuerySet Result, We can use the Logical Operation like AND(&) and OR(|) **from Django.db.models import Q**
* To do more complex comparison like greater than and lesser than and Startswith we can sue the FieldLookup Technique
* Syntax : <MODEL\_NAME>.objects.filter(<COLUMN\_NAME>\_\_<COMPARISON\_NAME>=<VALUE>)
* COMPARISON\_NAME are startswith,in,gte,lte and etc





* **Updating Models:**
  + If new column added, default value should be give in the model.py file or should give it while running makemigration cmd, then only migration will happen.
  + Default value can be null or any values for sure
  + Then run makemigrations and migrate CMD
* **Updating Entries**
  + Retrieve the Records which needs to update into variable
  + Update the required attributes under the Variables(object)
  + Call save() , this will update the DB



* **Delete Entries**
  + Retrieve the record which needs to be deleted, using get()
  + Call delete()



* **Connecting Template and Database Model** 
  + Two Major Features
    - Django Forms
      * Allow Django to automatically create forms from Python to template
    - Class based View
      * Automatically generates View based on a Model
  + These feature we can learn in upcoming sections
  + Just import the models in the view.py
  + Do Db operation and get the Data from DB
  + Pass the Data as need to the Template in the context attributes in render method
  + In template , use Django expression to display the value

**SECTION FIVE (my\_car\_site)**

**Django Admin Section**

* Django will be come with Admin Interface under the website manager
* We already have “/admin” in the urls.py in the Project scoped Files
* And also it is already configured in the setting.py file of INSTALLED\_APPS.
* **IMPORTANT :** This admin portal is used by the Website Owner/Administrator, it is not for the normal user login purpose

**Model With Website**

* {% csrf\_token %} add this inside the html form of the Django template to avoid the csfr error
* According to Django, One endpoint will act as GET request when it loads the page(When user land into that page), When user submit or refresh the page it consider as POST request to the same end point(the end point which used for rendering the page)

**Accessing Django Admin Portal**

* We can access admin portal by <OUR\_DOMAIN\_NAME>/admin
* Create the “superuser” via cli cmd to access the portal.
  + python manage.py createsuperuser
* Use this credentials and login into the portal

**Connecting Models with Admin Portal**

* Open the admin.py file in the App scoped file
* Import the models which is needed to be controlled by the admin portal
* Use admin.site.register(<Model Name>)
* Now login into the admin portal and you can see your Model in the admin portal
* Once you go inside of the Model, you can view the string representation of an each stored objects.
* We can click on the each object(Each record) and perform the CURD operations on the record.
* IN this way we have only limited access to the model in the Admin Portal.
* To overcome this issue and get many control over the model can be done by extending the our model with **model.ModeAdmin.**
* Create a class and inherit the ModelAdmin and pass this class object into the register method. admin.site.register(<Model Name>,<Class\_Name>).
* By this method we can change the oder of the field.

**SECTION SIX(Django\_forms)**

GET,POST and CSRF

**Django Form Class View**

* Two things are must in the template form tag and submit button within the form tag, apart from this we wont use any html tags
* Create a python file in the App Scoped Folder
* Define a class inside that file and inherit the form class of Django as below
  + from django import forms
  + class <OUR\_CLASS\_NAME>(forms.Form)
* This class is called as logical structure of an HTML form, it make us to keep template clean
* Import this class into the view.py
* Create an object for this form and pass this form in context to the template
* In the template use {{<CONTEXT\_VARIABLE>}}

**Template rendering**

* By adding **as\_p**  in the template tells the Django to put <p> in each elements
* We can add as **as\_ul** and **as\_table**
* We can able to access each element in the CONTEXT variable to give any unique props value for it.

**Form Fields and Widgets and CSS**

* Refer widgets\_validation\_styling.html

**ModelForm and Validation**

* Refer Official Documents

**SECTION SEVEN (class\_based\_view)**

**Class Based Views:**

**Generic Views – Template Views**

* Import : from django.views.generic import TemplateView
* Create Class and inherit TemplateView
* Create a variable template\_name=<TEMPLATE\_PATH>
* In urls.py, we import the class and use in the path()

**Generic Views – Form Views**

* Create FormView with your own fields , by inherit **forms.Form**
* Import : from django.views.generic import FormView
* Import The Form view in this View.py
* Here creare class and inherit FormView
* Create a variable form\_class=<FORM\_CLASS\_NAME>
* Create a html file(contact.html) and follow as instruction as **Django Form View**
* Create a variable template\_name=<TEMPLATE\_PATH>
* In urls.py, we import the class and use in the path()

**Generic Views – Create Views**

* This is Best View in the **Class Based View** section
* This is the **Model Based Class Based View**
* This is will create view for the end user by the **model class**
* **Important:** Template name should be in a pattern called <EXACT\_MODEL\_NAME>\_form.html
* Import : from django.views.generic import CreateView
* Create a class and inherit CreateView inside this
* Import models inside the view.py
* Create a variable **model=<MODEL\_NAME>**
* Create a variable **fields=”\_\_all\_\_”**
* Create a success\_url when the form success submit happens
* Create a template file in the naming pattern as said before(teacher\_form.html)
* Inside template value can be accessed using variable **“form”**

**Generic View – List View**

* Create a class and inherit ListView inside this
* Import models inside the view.py
* Create a variable **model=<MODEL\_NAME>**
* Create a template file in the naming pattern as <MODEL\_NAME>\_list.html(**teacher\_list.html)**
* Inside template value can be accessed using variable **“object\_list”**
* We can change the default name from **object\_list** to some other using **context\_object\_name** variable in this class
* Using **query\_set** variable we can write our own fetch queries like .all(), .get(), .order\_by(), .filter() and etc

**Generic View – Detail View**

* Create a class and inherit DetailView inside this
* Import models inside the view.py
* Create a variable **model=<MODEL\_NAME>**
* Create a template file in the naming pattern as <MODEL\_NAME>\_detail.html ( **teacher\_detail.html** )
* Inside template value can be accessed using variable {{<SMALL\_CASE\_OF\_MODEL\_NAME>}}
* It accepts the Primary Key as input and give the model object to the template.
* The path which accessing the this view should pass the path param as <int:pk>, so Django will help to get the single Model detail object.

**Generic View – Update View**

* Create a class and inherit UpdateView inside this
* Import models inside the view.py
* Create a variable **model=<MODEL\_NAME>**
* Create a template file in the naming pattern as <MODEL\_NAME>\_form.html ( **teacher\_form.html** ) or if it is already created in the CreateView Django will re-use of it
* Create a variable **fields=[<model\_variable\_name>],** if needed to control over the update operation in the specific columns
* Create a success\_url when the form success submit happens
* It accepts the Primary Key as input and give the model object to the template.
* The path which accessing the this view should pass the path param as <int:pk>, so Django will help to get the single Model detail object.

**Generic View - Delete View**

* Create a class and inherit DeleteView inside this
* Import models inside the view.py
* Create a variable **model=<MODEL\_NAME>**
* Create a template file in the naming pattern as <MODEL\_NAME>\_confirm\_delete.html ( **teacher\_confirm\_delete.html** )
* Inside template value can be accessed using variable {{<SMALL\_CASE\_OF\_MODEL\_NAME>}}
* Create a success\_url when the form success submit happens
* It accepts the Primary Key as input and give the model object to the template.
* The path which accessing the this view should pass the path param as <int:pk>, so Django will help to get the single Model detail object.