**Django**

**Basics**

Before you start with your Django project, it is better to create a new virtual environment for that project. It allows you to work with dependencies and shield your application from backwards compatibility issues. After creating the virtual environment, install Django in that environment.

Activate the virtual environment and go to your project folder. Django adds certain environment variables to your OS while installing. These variables are used while working on a project.

To start a new project navigate to the folder your virtual environment is located in, and type the command ‘django-admin startproject project\_name’. A folder with the project name will appear, it will contain a folder with the same name as the project, and a ‘manage.py’ file.

The subfolder contains certain python files that mean following:

\_\_init\_\_:- A blank python file that tells python interpreter that this folder can be treated as a package.

settings:- A file to store all the project settings.

urls:- Stores all the URL patterns of your project. Basically all the webpage addresses.

wsgi:- Web Server Gateway Interface helps us to deploy our app to production.

asgi:- Asynchronous Server Gateway Interface

The manage.py file is Django's command-line utility for administrative tasks.

To start the Django inbuilt development server type ‘python manage.py runserver’ from your project directory. This will return a local link, which if you copy and paste on to your browsers URL box will show your project.

**Initializing an App**

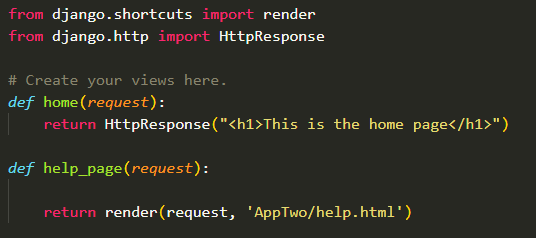
Django allows you to create several webapps inside of your project. Apps that can add various functionality to your project. You can add apps created by other developers into your project as well, this is something known as pluggability.

To create a new app run ‘python manage.py startapp app\_name’ from your project folder. Do not forget to add your apps name manually as a string, in the INSTALLED\_APPS list inside of settings.py file of the main project.

Your app will have its models, url mappers, views, forms and many other functionalities.

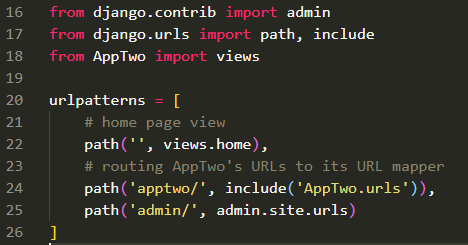
**Views & URL Mapping**

Inside of your app folder, you will find a views.py file. This file should contain all the views that this particular app will present. Each view is defined as a Python function or a Python class. The function/class name is the name of that particular view. A view takes an instance of HttpRequest as an argument. When a user does something, the browser sends an HTTP request to the server. Django takes the URL pattern and the HTTP request, it goes through all the url patterns mentioned in the urls.py and finds a corresponding view for that pattern. It implicitly creates an HttpRequest object defined in django.http and passes it to the view. Django views define what to do with these requests, therefore they need to take a HttpRequest as an argument.



A view is defined as a callable That takes a request and returns a response, it is usually a function or a class with .as\_view() method

Once the views are well defined, we need to add these views to a URL mapper. In Django URL mappers are urls.py files that can be created inside of app folders. However, the root URL mapper is located in the project configuration folder. To add our views go to urls.py file of the project file, there you will find all the necessary imports already done for you, but you still need to import your views.py file to his module. As our app folder has \_\_init\_\_.py file it is treated as a package, use ‘from app\_name import views’ to import the views.py file of the app. Inside of urls.py, you will find a list of URL patterns that takes URL pattern as a string and the view to trigger when Django finds that pattern in the request URL.



Let us assume you have a site named ‘xyz.com’ and you want to create a help page. You want Django to trigger a help view on URL pattern ‘xyz.com/help/’. In order to do this, insert the following into the urlpatterns list

‘path(‘help/’, views.help)’.

As soon as Django comes across ‘help/’ in the URL it will trigger the help view. Set the URL pattern string empty to for the home page.

If your project is going to have multiple apps, it is a lousy idea to store all the possible URL patterns to the root urls.py file. In such cases you should rout the app specific URLs to the apps own urls.py URL mapper. I order to do this go to your apps main folder and create a urls.py file, import the same modules as in root urls.py file, create a list named ‘urlpatterns’ and populate them with path objects, as in the root URL mapper. In the root URL mapper give a URL pattern string for the app and in the place of views add a include method, and pass the apps urls.py module as a string, as shown in the image above (line-24) this will route any URL with pattern apptwo/ to apps URL mapper. Lets assume your app has a home view and a user view, then the URL ‘xyz.com/app/’ will trigger apps home view and ‘xyz.com/app/user/’ will trigger user view. All the app specific URLs will be in pattern ‘app/xyx/’, for defining apps home page put a blank string in apps urlpatterns list and give a home view.

For in-Depth Information, refer docs.

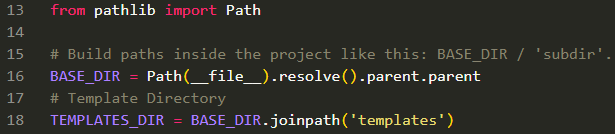
Views Doc: <https://docs.djangoproject.com/en/3.1/topics/http/views/>

URL [https:/docs.djangoproject.com/en/3.1/topics/http/urls/](https://docs.djangoproject.com/en/3.1/topics/http/urls/)

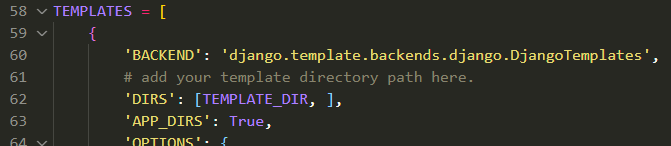
**Templates**

Django works on a Model View Template Design pattern, where the Model represents the database through an ORM model and View is the intermediary that takes requests and returns the rendered HTML. Templates, is where all the front-end work is done, templates contain all the HTML files that will be rendered by the render function in the views.py file.

To get started with templates create a folder named ‘templates’ in your main project folder. It can be named anything, but it is better to give it an appropriate name. It is a good practice to create separate sub-directories for each of your apps inside the templates folder. Django needs to know where to look for templates in your project; therefore, you will need to add the templates directory path in your projects settings.py file. To do this go to settings.py and look for BASE\_DIR constant, this constant stores the path of your project in the OS. You cannot give path directly as a string, because different OS have different directory systems and file paths, therefore we use python’s pathlib modules Path class to define our directory paths. You can create Path instance for templates directory by calling .joinpath() method on BASE\_DIR constant, and passing it templates folder name, as shown below on line-18.

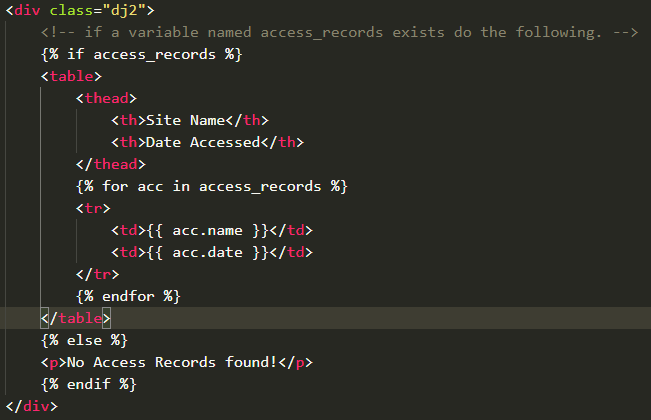


Do not worry about the apps subdirectory in the templates directory, as the views.py file and the browser handles it. Add this templates directory Path to the ‘DIRS’ list in the TEMPLATES list declared in the settings.py file, as shown below.



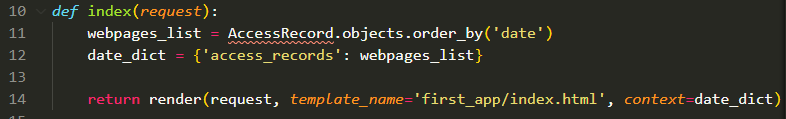
Your templates folder will contain all the HTML files. The way Django handles templates; is through its template tagging language that goes into the HTML file, and a context dictionary that contains all the objects that will be passed to the HTML. The context dictionary is a collection of key-value pairs where key is the variable name given in the HTML and value is the object being passed to it.

The way template tagging works is that you use double curly braces inside of which you place your variable {{ variable\_name }}. The same variable\_name should be a key in the context dictionary declared in the views.py, the value given to that variable in the dictionary will be placed inside the {{ }} in rendered HTML. For using logic inside of HTML use {% logic\_here %} template tags. Tags that use logic are needed to be ended, weather it is a loop or conditional statement. An example of using template tag is below.



The above template tag script checks if access\_record exists, if so creates a table and goes through each entry in access\_records. It takes name & date from each entry and puts them in table cells.

The view for rendering above HTML will be as follows…

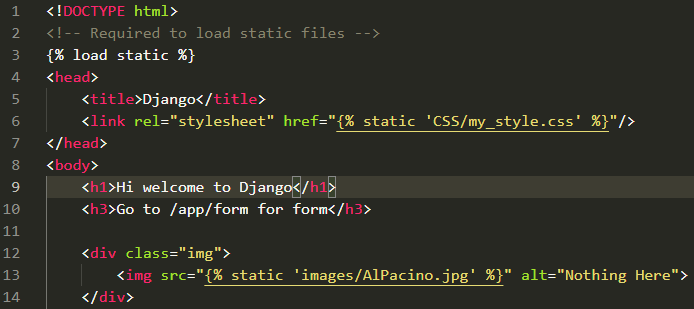


The view creates a list of all objects inside a model named ‘AccessRecord’, orders them by their date field and puts it in context dictionary named date\_dict, with a key ‘access\_records’ which is used in the HTML example above. The view returns a rendered HTML, where the render function takes the request it received, the template path and the context dictionary. Notice how we give the template path from inside the template folder, as Django will look for templates inside of TEMPLATES\_DIR Path that we passed it, this is where we need to give sub-directories.

**Static Files**

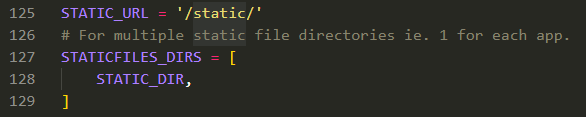
In Django static files are the files that are needed for our website, files like images, other media files, CSS files, etc. any file that will no be changed and is stored by the developer/admin goes here.

In order to work with static files create a directory for static files in your project folder and give it a name. It is good practice to create sub-directories for different types of files i.e. different directories for images, audios, CSS, JavaScript, etc. or, for different apps in your project. Make sure that django.contrib.staticfiles is included in your INSTALLED\_APPS in settings.py. Add a STATIC\_URL variable in settings and give it a string e.g. ‘/static/’. Inside of a template where you want these static files to be loaded, add a template tag {% load static %} at the beginning of the document. Use the template tag {% static ‘name/path of static file’ %} to build the URL for the given relative path to static file. An example for working with static files is below.



In the above example, we used static template tag to load a CSS file on line-6 and an image on line-13. The path given after the static keyword is from inside the STATIC\_DIR folder we declared in settings.py

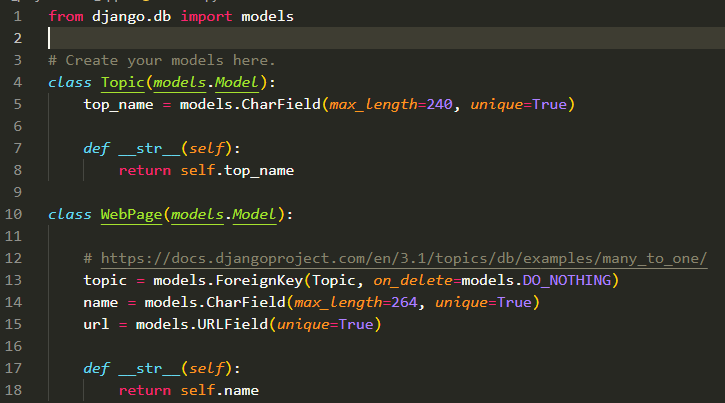
If your project has more than one static file folders, create a constant ‘STATICFILES\_DIRS’ in your settings.py file and give it a list of static file directory Path. Django will traverse these folders to search for static files. For in-depth info on static files read the documentation at <https://docs.djangoproject.com/en/3.1/howto/static-files/>



**Models**

In Django you there is no need to work around with SQL schemas for your database. Django uses something called an Object Relational Mapper (ORM) implicitly to define your data. What you do is, define a Model class that gives you an Object Oriented way of defining your Database; this model class represents a Table in your relational database. The class attributes represent a database field i.e. a table column. You can set the type of fields using Django’s in-built field types, a list of Django built-in field types can be found [here](https://docs.djangoproject.com/en/3.1/ref/models/fields/#field-types). The field types not only define the column types when creating a table but also the value for type parameter in HTML input tag while rendering an HTML form. An example of creating Model is shown below.

To create your model class you will need to inherit from Django Model class defined in Django.db.models.base. A large Models script can be broken down into smaller scripts and stored in models directory with a \_\_init\_\_.py file.

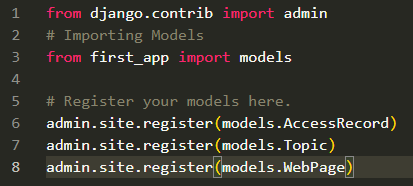


The above model will create 2 tables, Topic & WebPage in our database. The first model has a field named top\_name, which is a character field with 2 constraints. The \_\_str\_\_ method returns a string that is used to display an instance of that particular model in the admins database registry. On line-13 there is a field named topic, which is a foreign key to the primary key of Topic Model. You can set a primary key of a model by using the primary\_key kwarg of the fields methods. However, we have not assigned a primary-key column to Topic, so Django implicitly assigns a primary key column named ‘id’ to every model instance. The on\_delete kwarg defines what to do with the instances of this model; if the corresponding instance to which this is a foreign key, is deleted (i.e. what to do with all the instances of WebPage that have a particular instance of Topic as their Foreign-key is deleted). For a detailed explanation on on\_delete parameter check [this link](https://docs.djangoproject.com/en/3.1/ref/models/fields/#arguments).

Once you are finished creating models for your app, go to your project folder and open-up the command line. You need to make migrations before you can use the models, therefore run the commands in the given order

* python manage.py migrate
* python manage.py makemigrations app\_name
* python manage.py migrate

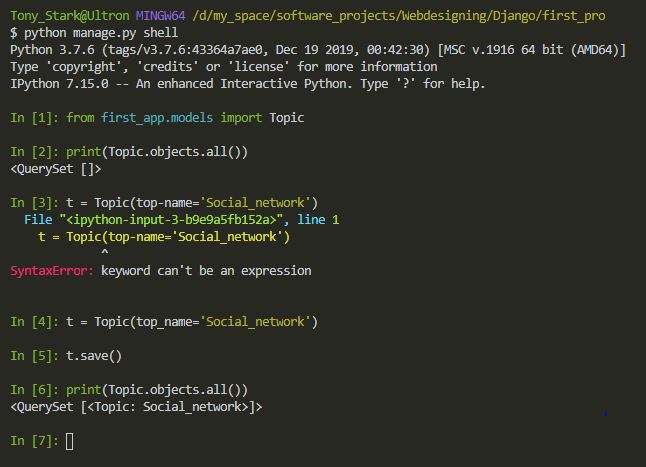
Django provides a Admin utility right out of the box. Model instances (Database entries) can be viewed graphically, from the admin interface, but in order to do that you need to create a super-user. To do that open-up the command line from project folder and pass the command ‘python manage.py createsuperuser’. Django will ask for your user-name, email and a password, save this information somewhere or if you forget the password you can use ‘python manage.py resetpassword’ command to reset password. After you create a super user register your Model in the admin.py file inside of your apps folder, to do this go to your apps admin file import the apps model module and use the .register method defined in Django’s admin module, as shown in the example below.



Once you are done registering your Models to admin run the migrations command again or you can do the Model creation and admin registration process at once and then run the migrations.

**Querying our database models**

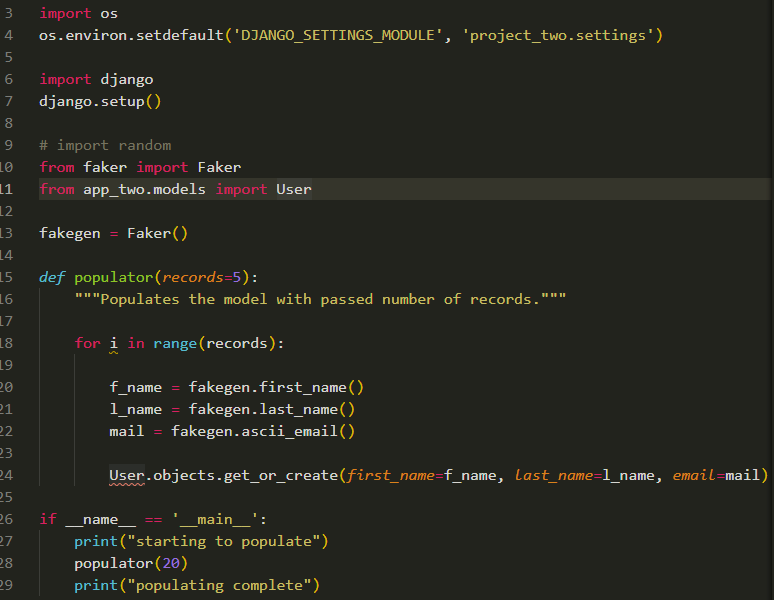
Once you have created your data models, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. You can use Django’s shell utility to test the Models. To start the shell go to your root folder and run the command ‘python manage.py shell’ this will start a python shell for the Django project. Import the models in the shell prompt and create a model object by passing it the necessary kwargs and use the .save() method on it to save it to database as shown in the example below. Each Model object represents a table row.



The Model.objects is a model manager. A model manager is an interface for a model to get its QuerySet object, it defines certain methods such as all(), filter() and exclude(). One can create their own managers in the app defined in manager.py file in app directory. QuerySet() object that is a like a list of instance of that particular model. It an iterable object that can be indexed. Creating an instance does not save its values into the database, to do that use the .save() method as shown on line-5, you need to call the .save() method every time you make any changes to the instance. For more in-depth information on querying checkout this [documentation](https://docs.djangoproject.com/en/3.1/topics/db/queries/).

**Fake Data Entries**

Often times a project might need large number of data inside of the database for testing purposes. You can use the faker module to add fake data into your database fields. To do so install faker module using pip. Create a script in your root folder for inserting data and inside of that script import your apps Models. Create a Faker() instance from the faker module. This Faker instance can be used to create all kinds of fake data like, phone numbers, addresses, URLs, Emails, credit card numbers, etc. To know more about faker check out this [link](https://zetcode.com/python/faker/) or this [link](https://faker.readthedocs.io/en/master/index.html). In the example below we use the os module to change the environment variable to point towards projects settings, then run Django’s setup method that is needed for logging and populating the app registry. After that we use Fakers methods inside a for loop to create fake data and also make use of Models .get\_or\_create() method that returns a list of objects with the passed parameter or creates one if none exists.

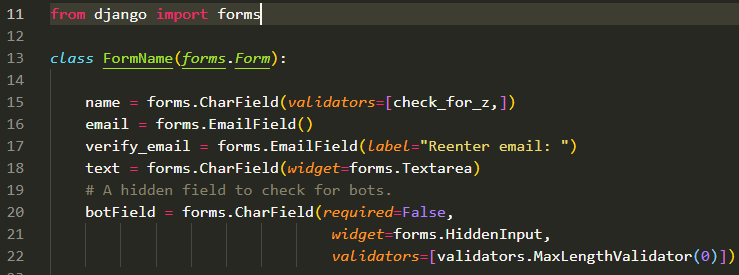


[**Django Forms**](https://docs.djangoproject.com/en/3.1/topics/forms/)

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

With Django, you do not need to create HTML form fields in your templates. All the form fields are defined in a forms.py file in your apps directory, by an instance of Django Form class defined in its forms module.

To create a Django form, make a new python module inside of your apps directory named forms.py. Inside this module import Django’s forms module (line-11). Create a class and give it a appropriate name according to your form and make it inherit from forms.Form class (line-13), inside of this class you can define form fields that are based on pre-defined classes in the forms module, for a full list of pre-defined fields checkout [this link](https://docs.djangoproject.com/en/3.1/ref/forms/fields/#built-in-field-classes). These fields can take field validators, constraints, widgets, etc as keyword arguments that can determine the rendering & behavior of that field. For a full list of these parameters checkout [this link](https://docs.djangoproject.com/en/3.1/ref/forms/fields/#core-field-arguments).

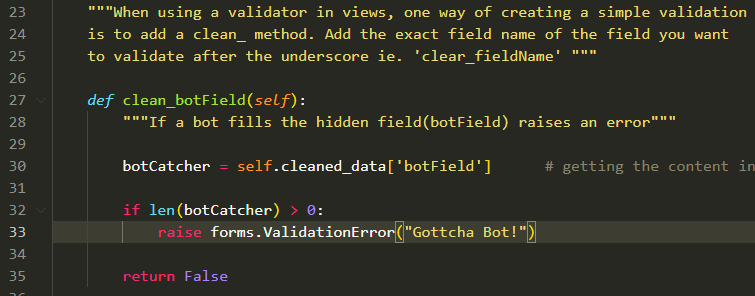


In the above example we create a Form sub-class named FormName that has four visible fields and one invisible field to check for bots. The widget parameter on line 18 & 21 determined how the form field will be rendered and it overrides the default render of Charfield().

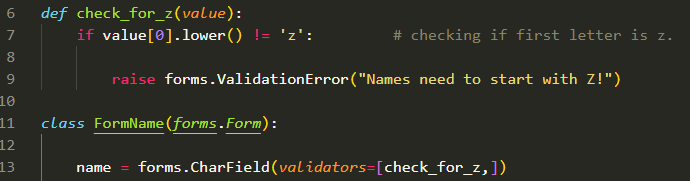
**Form validators**

If you are going to provide forms on your website, you will also need to provide some sort of validation to the fields. Basic validations like email validation, IP address validation, phone number validation are carried out by default depending on the field type you have defined in your form module. However, if you want to add more validation checks you can use Django’s built- in validators or define your own validators inside of forms module and pass it to the validators’ parameter of that field. To use built-in validators import validators module from django.core and pass the appropriate validator in the validators parameter as a list element, as shown on line-22 in the example above. For a detailed list on built-in validators checkout [this link](https://docs.djangoproject.com/en/3.1/ref/validators/#module-django.core.validators).

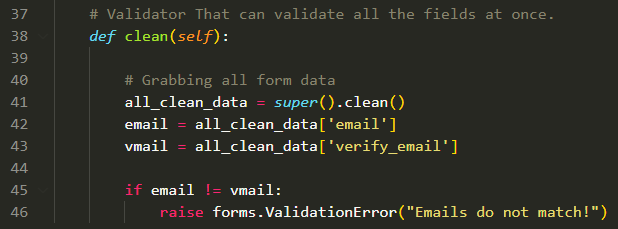
You can define custom validators as a class method or as an externally defined function inside the forms module. There are two types of custom validators, one that checks single fields and one that can check all field at once. You can define a single field validator inside of your form class by using the clean\_ method. Define a method whose name starts with clean\_ followed by the field name you want to validate and define the validation logic inside the method. Make it raise an error and pass a Boolean False if the data is invalid.



Here, is an example of externally defined validator



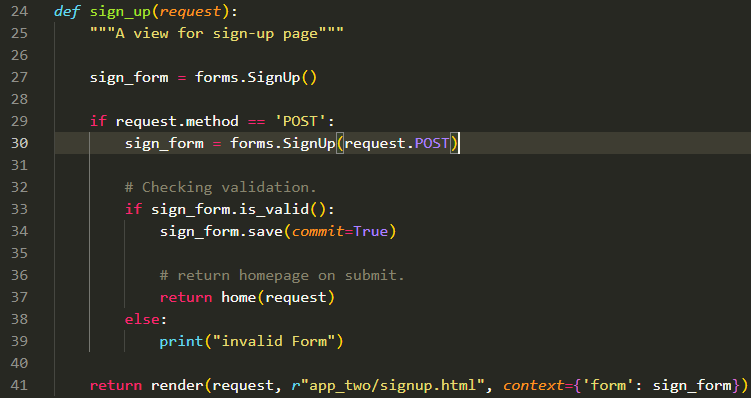
The difference between externally and internally defined validator is that you need to pass external validators to the validator’s parameter in fields. Where else internal validators are bound to Forms .clean() method that checks for validation when Form.is\_valid() method is called from the view and that is why you need to use the clean key word in some form while defining them.



Above is an example of a multi-field validator that checks for both emails to be same. The super().clean() returns the data in all the fields as a dictionary.

In order to render our forms in our HTML we need to create a view for them. To do that import forms into views module and create a form view function that takes a request as a parameter (Generally an HTTP request). Create an instance of the form and pass it in the context dictionary with an appropriate string key that you will use in the HTML template.

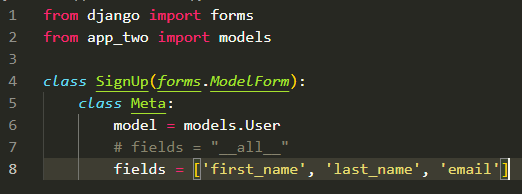
When a user fills the form and pushes the submit button, the browser sends a HTTP request with a POST method (in django). The validation functions must be triggered after the submission and not when the page is rendered, therefore use the following code to validate and further process the form.



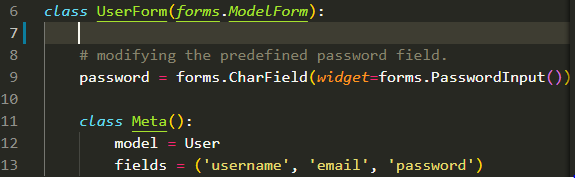
In the code above a form instance is created on line-27, and it is rendered with the signup.html as the user has not yet filled the form and pressed submit. After filling the form and pressing submit we receive a POST request that we validate on line-29 by using request.method attribute. Create a new form instance and check if it is valid using the .is\_valid() method that will run all the validation function defined in the forms.py file. If it is valid, it returns the home view i.e. it redirects the user to the home page.

**Model Forms**

If you are building a database-driven app, chances are you will have forms that map closely to Django models. For instance, you might have a BlogComment model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you have already defined the fields in your model. For this reason, Django provides a helper class that lets you create a Form class from a Django model called a ModelForm class. Below is an example of implementation of a ModelForm.



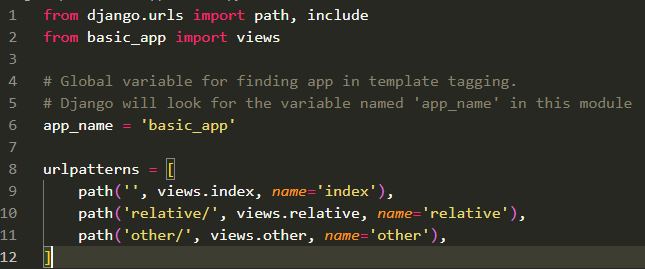
The generated form class from above code will have a form field for every model field specified in the model depending on the type of field. Each model field has a corresponding default form field. For example, a CharField on a model is represented as a CharField on a form. A ManyToManyField is represented as a MultipleChoiceField in the form. For a full list of conversions go to this [link](https://docs.djangoproject.com/en/3.1/topics/forms/modelforms/#field-types). The fields on the form will be ordered in the same order as they are represented in the fields list, use ‘\_all\_’ to take in all fields defined in the model. The ModelForm class should inherit from Django’s forms.ModelForm class and should contain a inline class named Meta, inside of which you mention the corresponding model and the field. If you want to modify a field from the model use the attribute name given to it in the model and assign it the desired form field and parameters. As shown below the password attribute defined in the User model is being given a different widget argument (line-9).

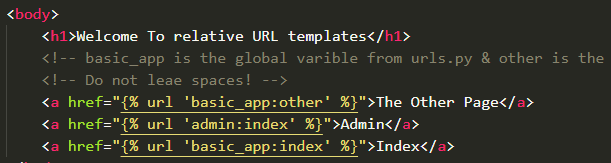


You can add form validations just like in normal forms; just create them outside the Meta inline-class. For more in-depth information on ModelForm look up this [documentation](https://docs.djangoproject.com/en/3.1/topics/forms/modelforms/#field-types).

**Relative URLs, template inheritance & filters**

When creating a webpage we often need to pass in lot of hyperlinks that we can pass through anchor tags <a> or in the ‘href’ attribute of some other tags. These hyperlinks can be of pages or views on your own website. Django provides convenient ways to pass URLs in your templates. A way to create relative URLs is to add an additional parameter to the path object in the urlspatterns list in your apps urls module. Add a name parameter to the path and set a variable named ‘app\_name’ to a string representing your apps name. The name parameter will be used to address the view from the template itself. Below are example of views and templates for using relative URLs





In the templates we make the use of template tags {% %} inside of which we use the url keyword that tells Django that this is a URL, after that we pass-in something called a namespaced URL. Namespaced URLs are specified using the ':' operator. For example, the main index page of the admin application is referenced using 'admin:index'. This indicates a namespace of 'admin', and a named URL of 'index'.

Namespaces can also be nested. The named URL 'sports:polls:index' would look for a pattern named 'index' in the namespace 'polls' that is itself defined within the top-level namespace 'sports'. The first part in our namespaced URL is the app\_name we provided in our urls module and the second part is the name parameter we provided to the URL we want to insert in that anchor tag, for more in depth information checkout this [documentation](https://docs.djangoproject.com/en/3.1/topics/http/urls/#url-namespaces).

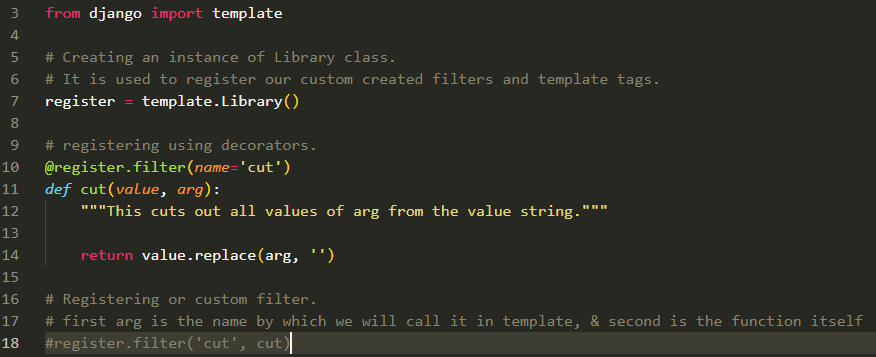
**Template Inheritance**

When building a website, you might want to keep certain features universal; across the website or at least across some sections of your website. In such cases template inheritance can help you a lot. Template inheritance allows you to define a base html that contains all the common elements of your site and allows you to define blocks, that child templates can override. For a detailed explanation of template inheritance checkout this [documentation](https://docs.djangoproject.com/en/3.1/ref/templates/language/#template-inheritancehttps://docs.djangoproject.com/en/3.1/ref/templates/language/).

**Template Filters**

Template filters can be used to modify the information shown by a variable on a webpage. Generally to add a variable in an HTML page we make use of {{ variable\_name }}, to add filters to it use the | operator followed by the filter name, followed by : and arguments if any, e.g. {{ k|add:’2’ }}, this built-in filter will add 2 to any value passed to the variable k. There are plenty of predefined template filter in Django listed [here](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#built-in-filter-reference).

To define your own filters go to your apps directory; add a templatetags folder inside of which you can create module with any name. Inside this module import Django’s template module, create a .Library() instance. This instance can be used to register filters or even your own custom template tags. An example of creating and registering filters is shown below.



The filter function above, takes the variable in the template on which it is applied as value, and an argument and removes the argument from the variable if there in the string. To register the filter we can use the direct method as on line-18 or through using decorators as on line-10. Remember to use the name you provided while registering and not the function name.

Below is an example of using this filter in a template.



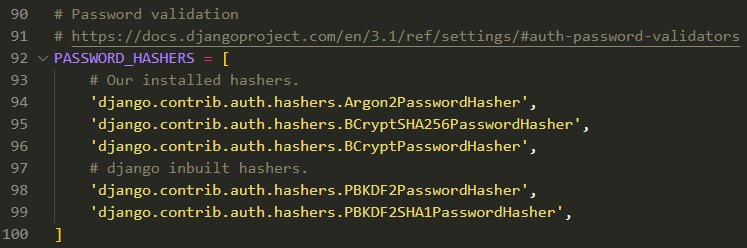
The code above will apply he cut filter on text and remove the string ‘hello’ if it is exits in the string passed to text.

**Working with Users**

Many website require a User registration and login to view certain contents, or even to use certain features. Django provides features to create apps that work with user registration & authentication.

**Password Hashers**

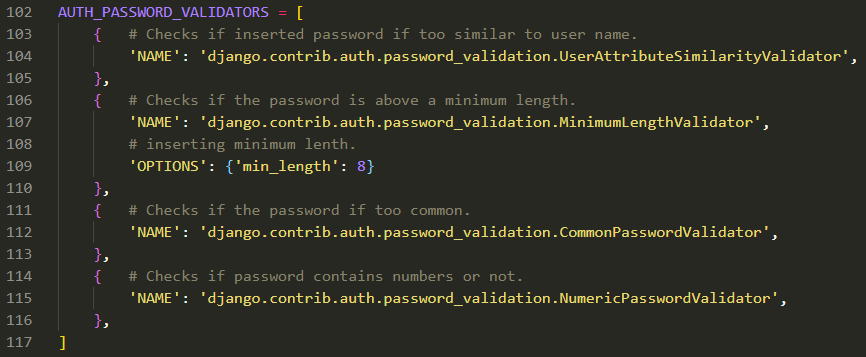
If you have users registering to your website and submitting a password to access their account, you must not save their password in plain text. Django by default encrypts the passwords, using the PBKDF2 password hasher if you use the inbuilt User model to create user models. However, you can add your own password hashers or install third-party hashers using pip. Make sure you have django.contrib.auth and django.contrib.contenttypes installed in your INSTALLED\_APPS list in settings.py file, If not add them and run migrations. Activate your virtual environment and run these pip commands ‘pip install bcrypt’ and ‘pip install django[argon2]’. These commands will install these hashers that will encrypt the password. Once installed you will need to mention the password hashers in your settings.py file. To do so, go to settings.py file and search or ‘# Password validation’ comment, under that comment add an identifier by the name PASSWORD\_HASHERS and give it a list of strings that mention the location of the hashers. Usually, you can find these hashers in django/contrib/auth directories hashers.py file. Add the hashers in the order you want them to be used, below is an example.



On the above example Django will try to encrypt using the first hasher mentioned i.e. Argon2PasswordHasher, and if for some reason Django could not do it, it will move on to the next hasher and so on. Therefore, it is a good practice to add hashers in the order of their reducing complexity.

**Password Validators**

Django adds certain password validators beforehand in the AUTH\_PASSWORD\_VALIDATORS list. Inside this list, you will find a list of dictionaries, each dictionary containing a ‘NAME’ key with the location of the validator as value. Some common validators used are below.



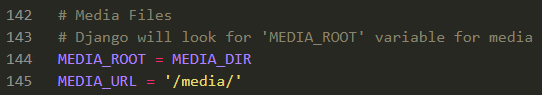
You can use the ‘OPTIONS’ key for a particular validator and pass it a dictionary of parameters that are accepted by that validator. You can find all the password validators in django/contrib/auth folders password\_validation.py file, you can add your own validation methods in this file and add them to the AUTH\_PASSWORD\_VALIDATORS list in the settings.py file.

**User uploaded media files**

Users on your site will upload certain media files such as profile pics, pdf, text based documents, etc. to handle all this you need to declare a media directory path in your settings.py file. To do this add a directory named ‘media’ in your project’s directory and create a Path object for this directory in your settings.py file, as shown below.



Assign this path to an identifier ‘MEDIA\_ROOT’ where Django will search for media files. Also give a URL string and assign it to ‘MEDIA\_URL’ identifier, as shown below.

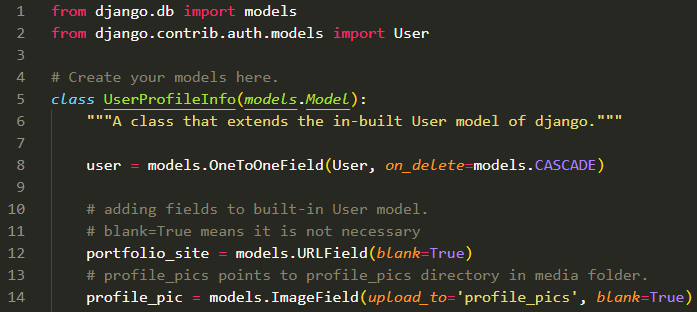


The user uploads files to already available form-fields, like a CommentField, a profile-pic form field, etc. Therefore, the input logic for these media files will be defined in the models for those forms. In order to work with image files in Django install pillow imaging package using ‘pip install pillow’ in your command prompt from your virtual environment. We will look at where to pass the path to the media file later while looking at models.

**Django’s Built-In User model**

To provide robust Authentication and authorization, Django comes with a Built-In User Model located in Django/contrib/auth folders models.py file. This Model represents users within the Django authentication system. This model contains default field attributes like username, email, password, last\_name & first\_name, username & password are required rest of them are optional. It also comes with Boolean attributes such as is\_active, is\_staff and date\_joined attribute.

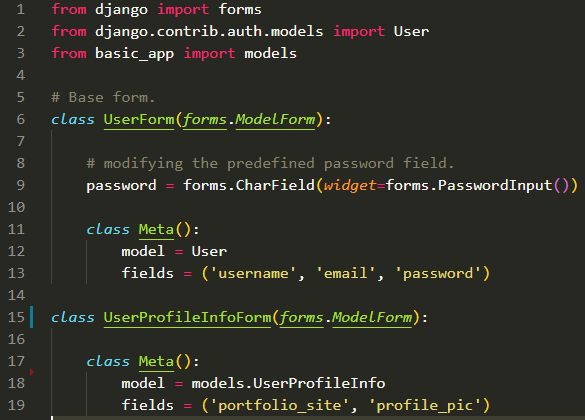
In order to create your own custom user model, you need to define a Model for your user in the models file of your app, and give a user attribute to it that has a OneToOneField relation with the built-In User model. We do not directly inherit the built-in User model as it is a sub-class of AbstractUser class and not the Model class, which our custom user model should be if we want to create a ModelForm from it; which we will do later on. An example of this is below.



In the above user model, we defined has a user attribute that has a OneToOneField relation with User built-in. the on\_delete method is set to CASCADE which will delete the instance of this model if the instance of built-in User is deleted. The built-in User model does not take URLs & images therefore, we defined additional attributes in our model for that. Here, the profile\_pic field is defined as an ImageField, what this does is that it stores the location of the image file that will be uploaded in the database and not the image itself. Therefore, we need to give it a location to store the User uploaded file. This particular model is asking for a profile picture, so we create a folder named ‘profile\_pics’ in the media directory and set the ‘upload\_to’ parameter to ‘profile\_pic’. Django will upload the picture to the MEDIA\_ROOT we provided in the settings by default. However, giving an upload\_to parameter will extend the path to MEDIA\_ROOT/profile\_pic. Once created register this model in admin.py file by importing them in admin module and using admin.site.register(Model) method.

**ModelForms for users**

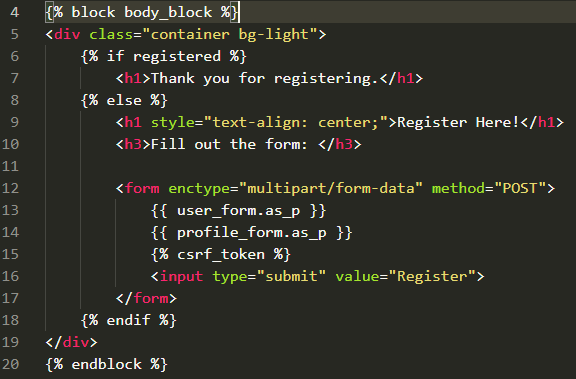
We know that the built-in User model and our custom user model are two different models, our model has an attribute that has one to one relation and the model itself does not inherit from the User model, we cannot create one ModelForm that can take our custom model as an attribute. What we can do is to create two separate forms for User and our custom user and merge them later in the view.



In the above example we import built-in User model and our model. We create a ModelForm for the built-in user, where we change its password fields default widget parameter. Then we pass username, email and password field, as those are the only fields we want. Next, we create a ModelForm for our custom user model, assign the model and mention the fields. These forms will get merged in the view file.

**The Templates**

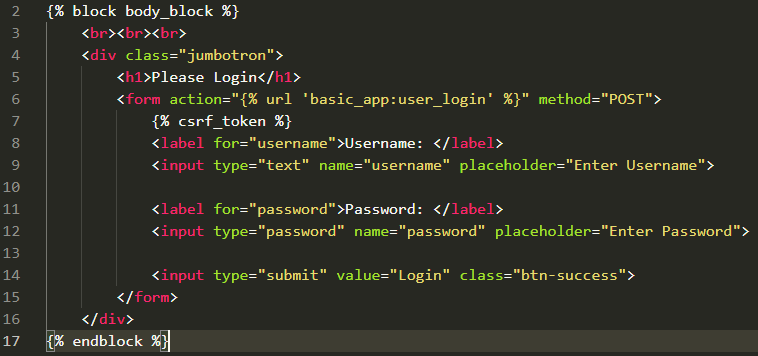
Before we going on to views, let us have a brief look at the templates we will need for user registration and user login. Given below is an example of User registration form.



In the above template, we check for a Boolean registered; if true (i.e. There is a user, who is registered but not logged-in and is viewing the register page) we pass a message. If the visitor is not registered, we pass a registration form, which is a combination of two forms, user form and profile form. Do not forget to pass csrf\_token on every form element. As our profile form is going to accept an image for the profile picture it is important to pass “multipart/form-data” to the ‘enctype’ argument of the form tag or else it will not accept the image.

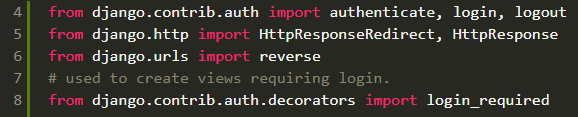
We have created neither a form, nor a model for our login, as there are no details required to be saved when a user logs in. The system just needs to authenticate certain parameters and if the are valid the user gets access to certain views. A login is a session, which allows the user to access certain views. for example, if you have a website where only logged-in users can make a comment, then you will need to create a form for the comment box, give it a view and use the @login\_required() decorator on that view. Like wise use such decorators on every view that requires a login. Below is an example of log-in form template.

In the form below we pass the relative url in the actions attribute of the form the routs the form to our log-in view, it asks the user for their username and password. The logging in process is carried out in the views file.



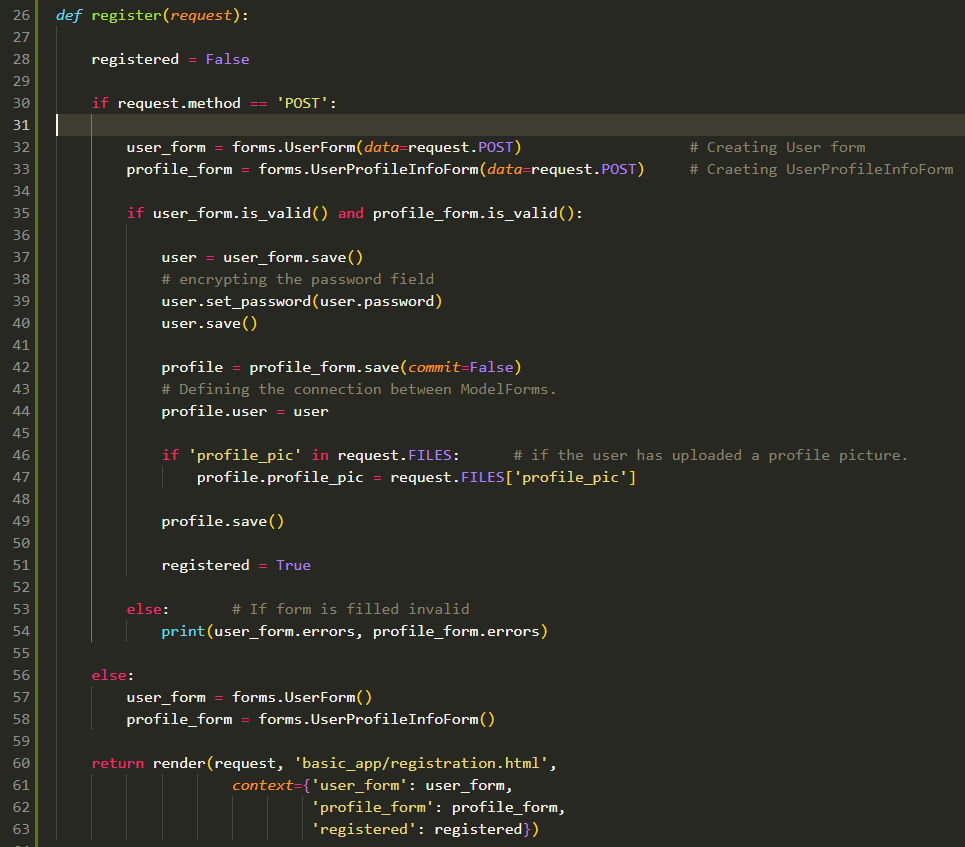
**The Views**

For users we need to focus on two type of views registration and login. There are many utilities in Django that will help us create these views. Apart from the usual imports you will need to make the imports shown below to add user registration, login and logout features.



Register View

The register view gets triggered when the user clicks on a register button on the home page. Given below is an example of a registration view.



In the above example, we first assume that the user is not registered and check if the http request method is ‘POST’. If so we create instances of our form with the data in POST, else we just create empty form instances and render them with the template. If the user submits information in the form, we check if the information in both the forms is valid on line-35. If not we print errors. If valid, we first save the user form to the database by calling the .save() method, and use the .set\_password() method passing in the password attribute to encrypt the password in the database, and save the user again.

We then go on to save the profile form, notice how we set commit=False on line-42, this is because we need to establish the relation between both the forms that exists in our model. Remember, there is only one model in our app, but two ModelForms. Our profile ModelForm is based on our defined model but the User ModelForm is based on the built-in User model. However, there is a user attribute in our model that has a one to one relation with built-in User. Therefore there must be a user attribute in our profile ModelForm that we can associate with the .save() instance we created on line-37, we do that on line-44.

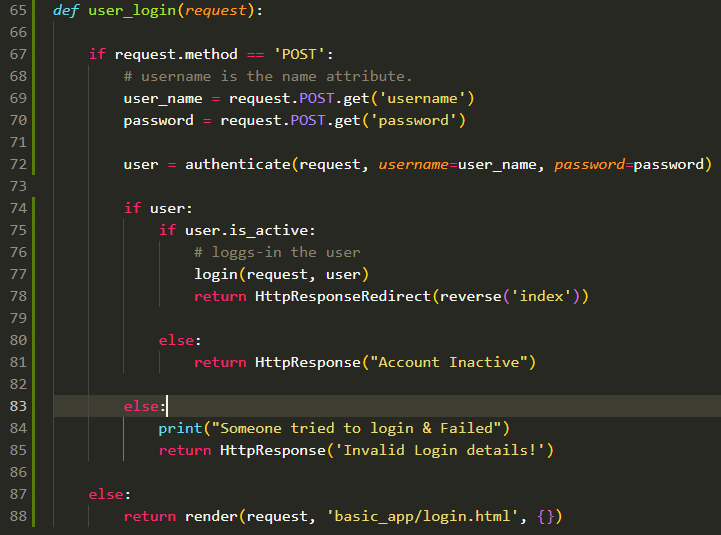
request.FILE returns a dictionary with HTML attributes as dictionary keys and the files uploaded by the user as values. We than check if there is a FILE with attribute ‘profile\_pic’, if so we assign it to profiles ‘profile\_pic’ field and go on to .save() the profile form. Now the user is registered and so the registered Boolean is set to True.

Login view

All the utilities we imported are needed to implement the user login system. Below is an example of a login view.

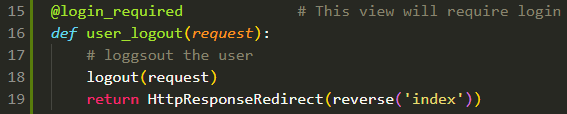
In the view below, we check if the request is a ‘POST’ request; if not we just render the blank login page for the user as is, if so get the values passed to the fields using the request.POST.get(attribute\_name) method, the attribute\_name is the string we passed to the name parameter in the forms input field. As we have previously mentioned all the user are represented by the Built-in User Model, if the person trying to login is already registered, There should be a username and a password for this saved in the database. We use the authenticate() method to see if the user is registered user or not; if not we return a response for invalid login, if so the authenticate method returns a User object and we assign it to user variable on line-72.

Before we login the user there is still one thing needed to be checked. The user.is\_active is a BooleanField that is set to True by default, however, it can be used to deactivate the user account, in case of which the user will not be able to login to his account. This Boolean field can be used to create systems such as, what to do if a user has not logged-in for years. You can give the user an option to deactivate his account in a profile\_view/profile options page. If the user is active we start his login session by using the login() method, it takes the HTTP request and the User object passed by the .authenticate() method. Once the user is loged-in we redirect him to the index page, the HttpResponseRedirect(URL) method is used to create a redirecting method that transfers the user to the passed URL. Here, instead of a URL we use the .reverse(‘view’) method that takes a view name as a string and creates a URL out of it.



Logout

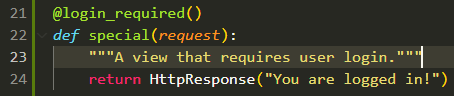
A logout mechanism is implemented using the Logout utility that we imported. Below is an example of a logout view.



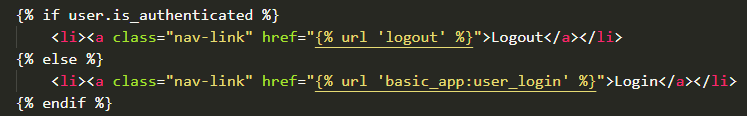
A logout will require a login in the first place, therefore, we use the @login\_required decorator to make this view available to logged-in users. The logout() method on line-18 ends the login session for the User. After that, we redirect the user to the home page.

Log-in specific views

In Django we define a lot of things in our views, things like how an HTML element will be rendered, what to do with an HTTP request, what to do with the model data, etc. Therefore, views are at the center of Django’s MVT architecture. We can make certain views to require a log-in by decorating them with a @login\_required() decorator. This decorator can be imported from django.contrib.auth.decorators module. An example of using this decorator is below.



A way to implement log-in check in templates is given below.



The user.is\_authenticated checks if the user is an authentic user; if so it will set the anchor to logout and if not it will set the anchor to login.

**Hosting on PythonAnywhere**

Many hosting service providers allow you to host your webapp on their servers. Most of them have a free tier option, through which you can deploy a small app and test it on a remote server. Pythonanywhere.com is one such service provider that specializes in providing hosting services for python based web apps. A thing to keep in mind is that different hosting service providers have different ways and methods to host your app. Generally it is on the following bases. The host will provide you with a remote machine with linux OS installed on it. You will need to install your dependencies on that OS, things like Python, Django & other software packages. Upload all the static files needed for the website to function, setup the virtual environment for the project setup a domain name and many other things depending on the scale of the project.

**PythonAnywhere**

Pythonanywhere is a web hosting service provider specifically designed for python based web apps. It provides features such as built-in support for python, IPython & PyPy, support DBMS such as MySQL & PostgreSQL, a built-in text-editor and console and a user-friendly interface for managing website.

The steps for deploying your web app are as follows

**Step 1: Signing up**

To deploy a project on pythonanywhere you will need to create an account with them. Go to their sign-up page and enter your Username, Email & password. With a free account you will get 512 MB of storage, 100s of CPU usage time that resets every 23hr, 2 consoles simultaneously, access to MySQL DB (No PostgreSQL) and a username.pythonanywhere.com domain name.

**Step 2: Git Repo**

The source code of your web app is cloned from a remote repository. Therefore, create a new repository of your web app on a git host.

**Step 3: Create a python virtual environment on server**

The virtual machine of pythonanywhere comes with python preinstalled. But if you want to run your application on a specific version of python, you can create a virtual environment with that version by using the following command:-

‘mkvirtualenv –-python=pythonx.x.x environment\_name’

To run this command, go to consoles tab and start a new Bash console.

As pythonanywhere’s server uses a linux OS, the virtual environment will be located at the following location:- ‘/home/user\_name/.virtualenvs/environment\_name’

**Step 4: install Django on the server’s virtual environment**

To install Django use the following command on the servers Bash shell:-

‘pip install –U django==x.x.x’

**Step 5: Clone your repo onto server**

To clone your remote repo, copy the git link from your remote git host. Type the following command on your servers Bash shell:-

‘git clone repo\_link’

The source codes will be located at the following location on the server:-

/home/user\_name/repo\_name/project\_name

**Step 6: Creating a Super user**

Create a super user on the server just like you would create on a local machine. Use the command after navigating to your project folder on server:-

‘python manage.py createsuperuser’

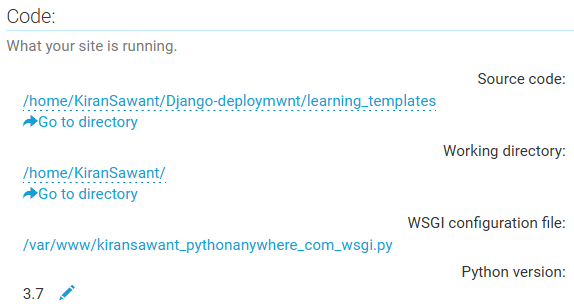
**Step 7: Run migrations**

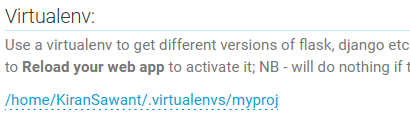
Run migrations by typing the following command from your projects root folder:-

‘python manage.py migrate’

**Step 8: Adding a Web Application**

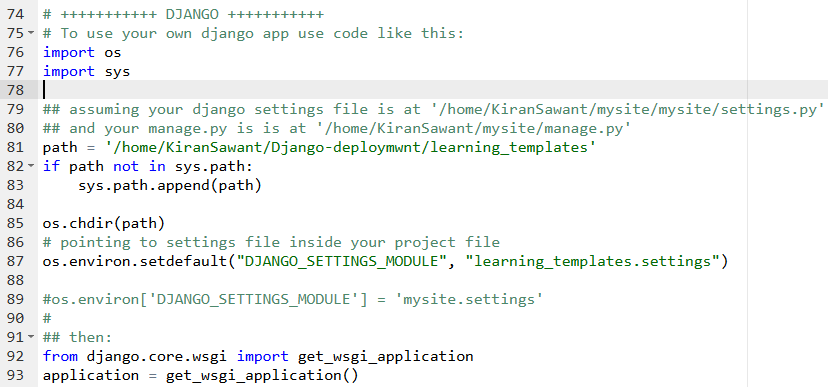
Go to the Web tab on pythonanywhere.com and press the add new web app button. It will take you to a series of forms that will ask you things about your web app. Things like project name, python version, framework, framework version, etc. However this does not give us the flexibility we want. Therefore, when selecting framework; select ‘Manual configuration’ option. This way we can mention our virtual environment that we created and use our repo that we cloned. Setting up manually requires us to configure our own WSGI file. Click next after selecting Manual setup. It will take you to a sample hello world page that contains a web app setup link, click on that link and go to the setup page. On the setup page you will need to provide path for various files and directories that your project needs.

**1.** Add the Source code & working directory file path in Code section. 

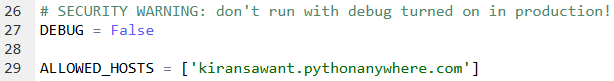
**2.** Add virtual environment path as in step-3 in Virtualenv section 

**3.** Configure the WSGI configuration file.

WSGI is the Web Server Gateway Interface. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. To get our web app running we need to manually configure our WSGI configuration file so that it can find project location and the projects settings file. We need to add our projects path in the server OS so that python knows where to look for our project specific modules, we do that using sys.pathappend(path :str). We set the “DJANGO\_SETTINGS\_MODULE” environment variable such that it points to our projects settings.py file, we do that using os.environ.setdefault(‘e\_var’, ‘path’) method as on line-87. We set the ‘application’ variable to a WSGI handler line-93. Do not forget to comment/delete the default Hello world code in WSGI config. Save the file.



**Step 9: Adding allowed hosts.**

You need to add the domain name of your website to the ALLOWED\_HOSTS list in your projects settings.py file as a string. Use the files tab in pythonanywhere to navigate to the settings.py file and add the domain name as a string.

**Step 10: Adding static files URLs & directory paths**

You need to add the path for your sites static files as well as Django’s static files that it needs for the Admin interface. Go to the static files section in the web tab on python anywhere and give the following arguments in the field.

**For Admin**

**URL**: /static/admin

**Directory**: /home/user\_name/.virtulenvs/env\_name/lib/pythonx.x/site-packages/django/contrib/admin/static/admin

**For Site**

**URL:** /static

**Directory:** /home/user\_name/repo\_name/project\_name/static

For security reasons, enable Force HTTPS.

**Class Based Views**

Class Based views were introduced in Django 1.4. CBVs allows for extra functionality to your view and the ability to define your views in an object oriented manner. You can leverage Python’s multiple inheritance functionality to make your views powerful without having to make them clunky and unreadable. Using CBV’s makes your code DRY compatible and allows for more reusability. For example, if you need a feed view to be present in two different views then you can create a general feed view and inherit them in those two views that need this feed view.

In Django, CBVs are implemented using something called ‘generic views’. Generic views are a pre-defined set of classes that can be inherited by our CBV to give it basic functionalities of a view. These generic view classes are located in Django.views.generic directory. A table of generic views is below.

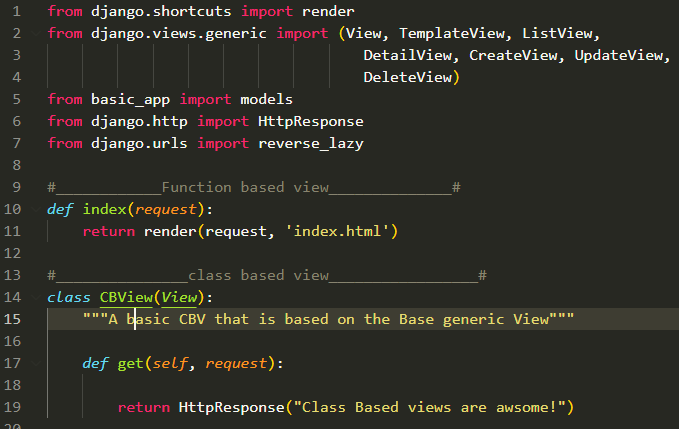
|  |  |  |
| --- | --- | --- |
| Type | Class Name | Description |
| Base | View | Parent of all CBV’s. Dispatching & Sanity checks. |
| Base | TemplateView | Rendering templates, connecting URL to context\_dictionary |
| Base | RedirectView | Redirects to another URL on GET Request |
| List | ListView | Renders Iterable of Items |
| Detail | DetailView | Renders field values of an item depending on pk or slug value. |
| Edit | FormView | Renders and processes a form. |
| Edit | CreateView | Renders and processes a form for creating an object. |
| Edit | UpdateView | Renders and processes a form for a pre-existing object |
| Edit | DeleteView | Processes a delete operation on an object and redirects. |
| Date | ArchiveIndexView | Renders a list of objects with a date field, latest being first. |
| Date | YearArchiveView | Renders a list of objects on a year given by URLConf. |
| Date | MonthArchiveView | Renders a list of object on a year & month. |
| Date | WeekArchiveView | Renders a list of object on a year & week. |
| Date | DayArchiveView | Renders a list of object on year, month, day. |
| Date | TodayArchiveView | Renders a list of object on this day. |
| Date | DateDetailView | Renders object detail on a year, month, day identified by pk or slug. |

**View**

The View class in the ‘base’ module is the parent of all generic CBV’s in Django. Other views inherit it, either directly or through multilevel inheritance. It performs basic sanity checks and dispatches. Other than that it also defines the .as\_view() method that is used in the URLConfig to pass it as a view. All CBVs must be passed with a .as\_view() method in the path() function in the urlpatterns list as shown below.



Below is an example of a basic class based view that sends an HttpResponse with its function based view counterpath.



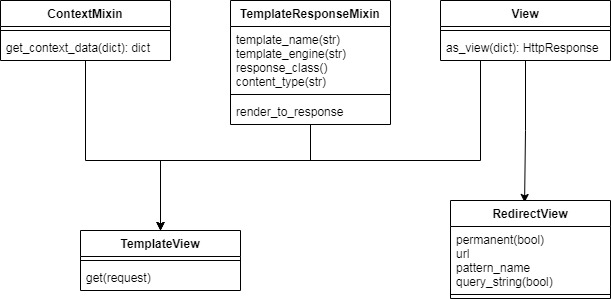
Get method is defined in View class and is used to return a response.

**TemplateView**

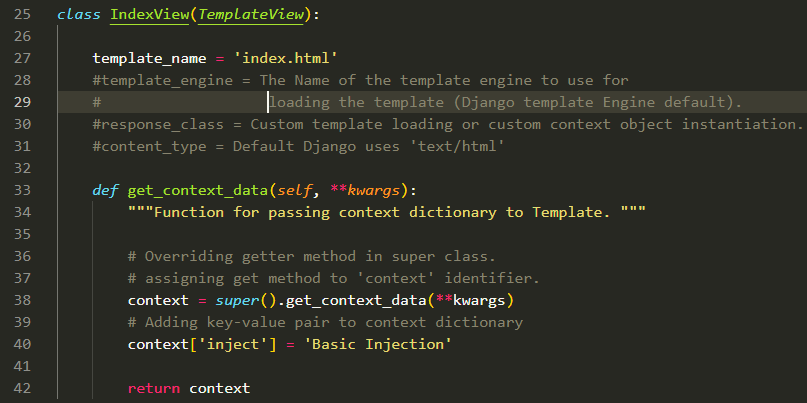
TemplateView is a basic generic view that is used to render templates. It does so using something called a mixin.

**Mixin:-** A mixin is a parentless class that takes advantage of pythons multiple inheritance to easily reuse chunks of code and functionalities. For example if you want to create a view that deals with rendering a template and defining a context dictionary, you can create functionalities for both the operations in two different classes and then inherit both of them in a child class and use it as a view. This is exactly what a TemplateView does; it inherits from TemplateResponseMixin for rendering templates, ContextMixin for creating context dictionaries and View for basic sanity checks.

As TemplateView inherits from all these views, it also shares all their attributes and functionalities. A few noteworthy functions are get\_context\_data() method that can be assigned to an attribute in your CBV using super() method, and then pass key-value pairs to that attribute just like a normal dictionary object. template\_name attribute expects a template path from your templates directory defined in settings. Below is a class diagram of the base module.



Below is an example of the implementation of a simple TemplateView.



The templateView requires a template\_name attribute that expects a template path as string from the TEMPLATES\_DIR path set in settings. There are other attributesthat can be overridden as shown rfom line-28 to line-31. We need to override the get\_context\_data method if we want to pass a context dictionary. Generally, TemplateView is not used directly, but rather as a mixin. However, we can use it If we just want to render a static page, like an about page or a help page.

**RedirectView**

Redirects to a given URL or a pattern. The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is always done (even if no arguments are passed in), any "%" characters in the URL must be written as "%%" so that Python will convert them to a single percent sign on output. If the given URL is None, Django will return an HttpResponseGone (410).

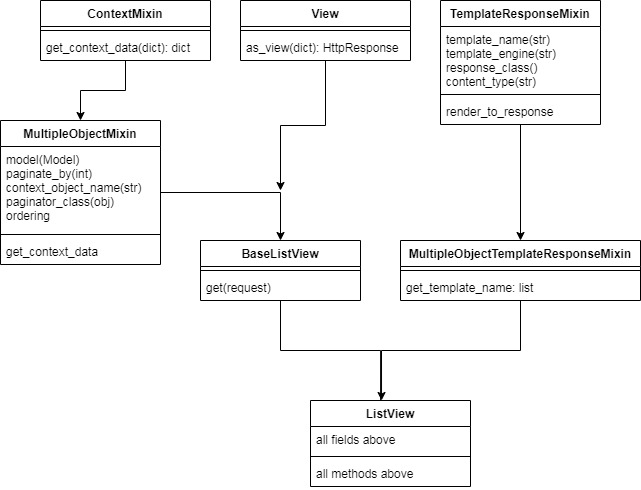
Below is an example of a redirect view. It redirects the user to a DetailView for a blog. This RedirectView sits in the middle of a ListView and DetailView, and is used to count the number of clicks a post has. get\_redirect\_url() method returns the URL to redirect to with the kwargs, but before doing so it perform certain actions like manipulating the counts field in the Post model for that particular instance. The get\_redirect\_url() uses pattern\_name to create redirect URL



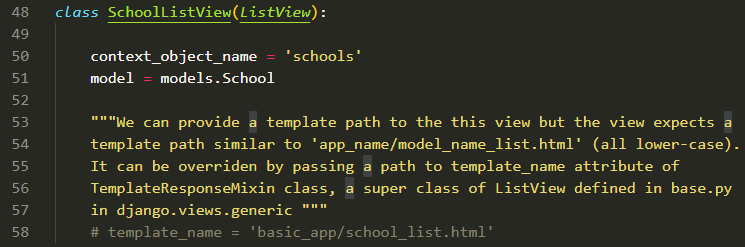
**ListView**

ListView is used to render an iterable list like view of instances of a model. For example in a blogging website, a list of cards that contain a snippet of the blog, name of the author and date published. You can use this view in several different ways, like list of blogs by an author, or list of blogs written today, or list of blogs belonging to a certain topic, etc. ListViews can return full details about the instances. However, it is discouraged doing so, and there is a separate view for it (DetailView).

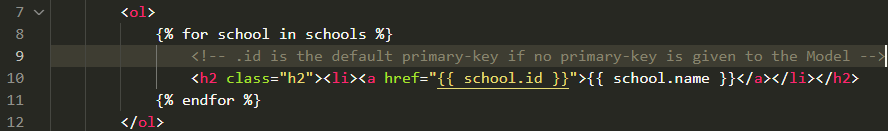
A ListView requires a Model, assigned to the model attribute. context\_object\_name attribute takes a string that is used in the HTML template as an iterable to iterate over. ListView inherits from MultipleObjectTemplateResponseMixin, which in turn inherits from TemplateResponseMixin, and therefore a ListView has a template\_name attribute that can take a template path. However, by default the ListView expects a template path similar to ‘app\_name/model\_name\_list.html’ (all lowercase). A thing to note though is that this path is not from the templates directory mentioned in the settings file, but from the templates directory in the apps folder. The way you should structure your directories is, make a ‘templates’ directory in apps directory and make another directory in it named as the app, inside of which you should store all the templates specific to the app. This behavior can be changed by simply overriding template\_name attribute. Below is a class diagram of list view.



Below is an example of implementing a ListView. The ListView takes a Model and passes a context dictionary to the template. The dictionary can be iterated over in the Template to display a list of instances of the Model. The dictionary is refered to by the context\_object\_name that we give it, in this case ‘schools’. We can add functionalities like pagination using ListView.



Below is the template for the view above

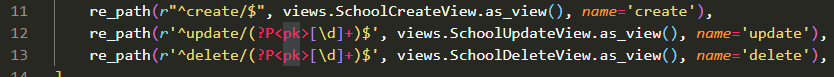


The template above will display a list of school names that can act as hyperlinks. We pass the primary key of that particular instance to href and then the URLConfig will handle the rest. Generally, URLConfig will look at the pattern and pass the id as pk to a DetailView that will then return a rendered detail template.

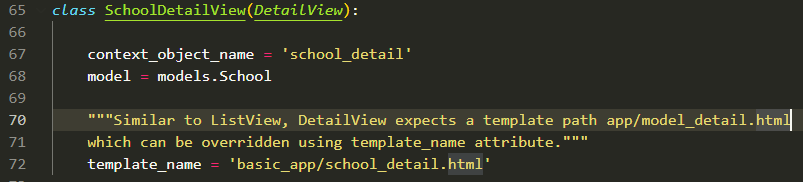
**DetailView**

A detail view displays details of a specific instance of a Model. It takes an argument ‘pk’ for primary key from the URLConfig. The hrefs in the templates should be created such that they pass the primary-key or a slug-field for the instance that is needed to be retrieved. An example of Detailview is to display the entire blog, when a user clicks on a summary blog on his feed. Similar to ListView; DetailView has a context\_object\_name attribute, model attribute and template\_name attribute. For template\_name it expects a path similar to ‘app\_name/modelname\_detail’ which can be overridden. The difference between a ListView and a DetailView is that; ListView is designed to render limited information of many instances, where else DetailView is designed to render the entire detail of a single instance.

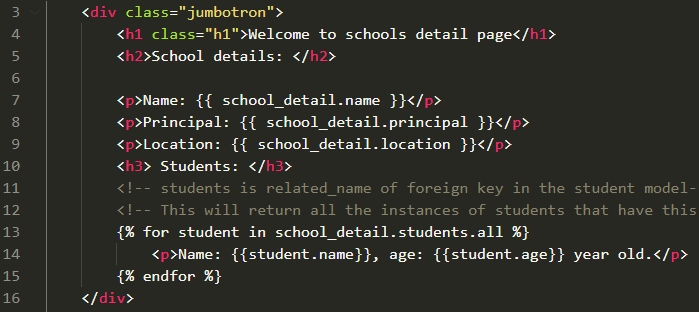
Therefore, DetailView requires an identifier of the instance to pass the details. The identifier is generally a primary key attribute ‘pk’, or a slug field. When calling DetailView from URLConfig you should use an re\_path() function instead of the standard path() function and capture the identifier field in a group represented by parenthesis ( ). The syntax for capturing the identifier looks similar to this (?P<pk>\d+), Here, URLConfig will pass a kwarg with key ‘pk’ and value captured in the group to the DetailView (slug can be used as well), ‘\d+’ is a regular expression expecting 1 or more digits. To memorize the syntax, use mnemonic “parents question pink action-figures”, below is an example uf passing a DetailView in URLConfig.



Here, if the URL has a pattern “update/5”, the URLConfig will use the SchoolUpdateView and pass it a kwarg ‘pk=5’, The DetailView will look for an instance with primary-key 5 in the model passed to it. It will return a render response as defined in it. An example is shown below.

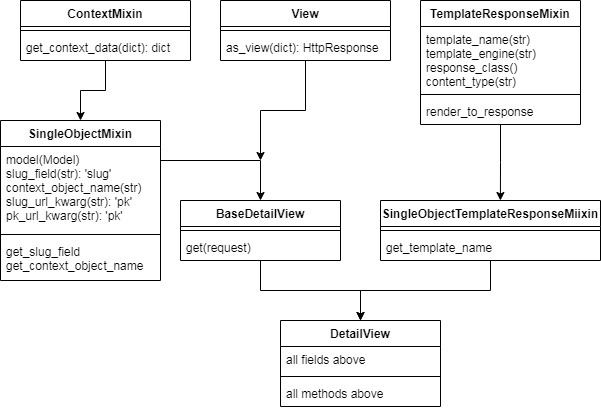


This is how the template for this view will look like.



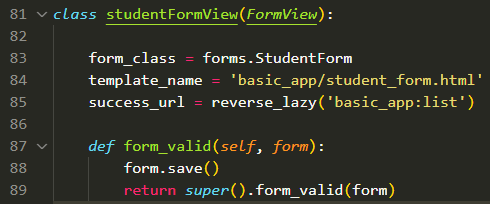
The Model that we passed to the DetailView has no students attribute. However, there is another Model; students, that has a school attribute which is a foreign-key to Schools Model and therefore we can access the student instances that have this instance of School in their school attribute using related\_name kwarg of ForeignKey() method, we do that on line-13, using related name ‘students’.

For reference below is the class diagram of details module.



**FormView**

A FormView can be used to create a view that renders a form. FormView handles the rendering and validation process of a form. It requires a Form or a ModelForm object assigned to form\_class attribute, a template path string assigned to template\_name attribute and a success\_url to redirect to when the form is submitted successfully. Below is an example of a form view.



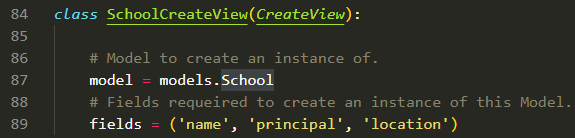
The FormView above is based on a StudentForm ModelForm. A FormView does not save the form data in the database, therefore we override the form\_valid() method that is executed when the form data is valid. We placed the form.save() method inside of it, so that if the form is valid; the form data is saved to the database.

**CreateView**

A CreateView is used to create an instance of a Model. For example, creating a blog post defined in a blog Model, or a user defined in User Model. It displays a form for creating an object, redisplays the form with validation errors if any, else saves the object on submit.

As it displays a form, it requires a form template. It expects a template path similar to ‘app\_name/model\_name\_form’, basically a form template followed by a ‘\_form’ suffix.

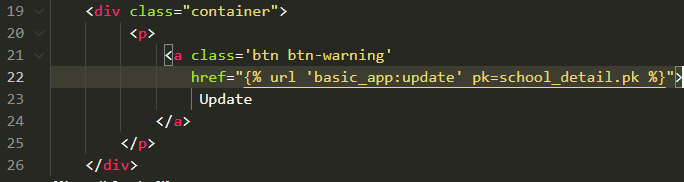
CreateView displays a form based on a Model, and therefore it needs to know the model and the fields of that model that are required to create that object. Pass the model to the model attribute and a tuple of fields to a ‘fields’ attribute, as shown below.



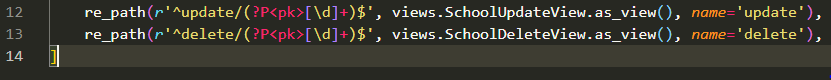
As soon as the user presses the submit button the object is saved to the database.

**UpdateView**

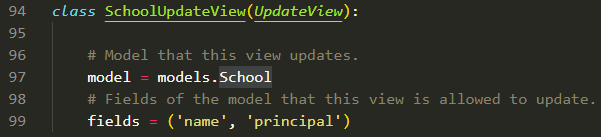
UpdateView is very similar to CreateView in the sense that it displays a form based on a model. The key difference is that it requires a pre-existing instance of the model and therefore requires an identifier of the instance that needs an update. This logic is handled in the template and the URLConfig file. Similar to DetailView it expects a ‘pk’ kwarg that is generally the primary key of the instance. Place an update button at an appropriate position like the detailView template, pass a url to the href that has a ‘namespace:pathname’ pattern and the kwarg that the path expects, as shown below.



In URLConfig it should be as follows



You can also pass a fields tuple that contains all the field names as string of the model that are allowed to be updated.



Code in URLConfig will be as follows.

Code in the template will be as follows.

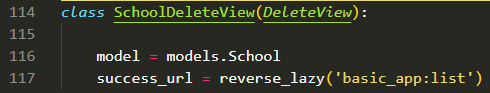
**DeleteView**

DeleteView is used to remove an instance of a Model from the database. While creating the delete view, you need set the model attribute to the Model of which the instance has to be deleted; as shown below. It also requires a success\_url attribute that redirects the user to a page after the object is deleted. The DeleteView expects a simple form template for confirmation, add a template in your apps template directory with a suffix ‘\_confirn\_delete’ preferable model\_confirm\_delete.

Place a delete button in an appropriate location, like in the DetailView template and pass it an href to the url that has a ‘namespace:pathname’ pattern and the kwarg that the path expects as shown below.



Below is an example of DeleteView.



In Python, functions are not executed on import; classes are. Therefore while using CBVs one should use reverse\_lazy() method from django.urls while assigning success\_url attribute outside a method, or else the URLs will be resolved the moment views module is imported to the urls module and not when a call for that particular view(SchoolDeleteView) is made, if so, the view will always return a wrong URL. reverse\_lazy() method resolves URL only when a call for that view is made and not when the module is imported. The URLConfig for DeleteView is shown above.

An example of confirmation template is below.

