How to configure Atom for Python:

1) Terminal Installation:

File->Settings->Install->in the searchbox just type terminal-->platform-ide-terminal

2) Python AutoCompletion:

File->Settings->Install->in the searchbox just type python-->autocomplete-python

3) django:

File->Settings->Install->in the searchbox just type djanngo-->atom-django

4) How to change terminal from powershell to normal command prompt:

File->Settings->Install->in the searchbox just type terminal-->platform-ide-terminal-->settings---->Shell Override

C:\Windows\System32\cmd.exe

DJango:

- Django is a free and open-source web framework.
- It is written in Python.
- It follows the Model-View-Template (MVT) architectural pattern.
- It is maintained by the Django Software Foundation (DSF)
- It is used by several top websites like Youtube,Google,Dropbox,Yahoo Maps, Mozilla,Instagram,Washington Times,Nasa and many more
- https://www.shuup.com/blog/25-of-the-most-popular-python-and-django-websites/
- Django was created in 2003 as an internal project at Lowrence Journal-World News Paper for their web development.
- The Original authors of Django Framework are: Adrian Holovaty, Simon Willison
- After Testing this framework with heavy traffics, Developers released for the public as open source framework on July 21st 2005.
- The Django was named in the memory of Guitarist Django Reinhardt.
- Official website: djangoproject.com

Top 5 Features of Django Framework:

Django was invented to meet fast-moving newsroom deadlines, while satisfying the tough requirements of experienced Web developers.

The following are main important features of Django

1) Fast:

Django was designed to help developers take applications from concept to completion as quickly as possible.

2) Fully loaded:

Django includes dozens of extras we can use to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds, and many more tasks.

3) Security:

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

4) Scalability:

Some of the busiest sites on the planet use Django's ability to quickly and flexibly scale to meet the heaviest traffic demands.

5) Versatile:

Companies, organizations and governments have used Django to build all sorts of things — from content management systems to social networks to scientific computing platforms.

Note:

1) As Django is specially designed web application framework, the most commonly required activities will takes care automatically by Django and Hence Developer's life will be simplified and we can develop applications very easily.

2) As Django invented at news paper,clear documentation is avilable including a sample polling application.

3) https://docs.djangoproject.com/en/2.1/contents/

How to install django:

1. Make sure Python is already installed in our system

```
python --version
```

2. Install django by using pip

```
pip install django
pip install django == 1.11.9
```

```
D:\>pip install django
Collecting django
  Downloading https://files.pythonhosted.org/packages/51/1a/
6153103322/Django-2.1-py3-none-any.whl (7.3MB) 100% || 7.3MB 47kB/s

Collecting pytz (from django)
  Downloading https://files.pythonhosted.org/packages/30/4e/
53b898779a/pytz-2018.5-py2.py3-none-any.whl (510kB)
    100% || 512kB 596kB/s

Installing collected packages: pytz, django
Successfully installed django-2.1 pytz-2018.5
You are using pip version 9.0.3, however version 18.0 is ava
You should consider upgrading via the 'python -m pip install
```

3. To check django version:

```
py -m django --version
```

Django Project vs Django Application:

A Django project is a collection of applications and configurations which forms a full web application.
Eg: Bank Project

A Dango Application is responsible to perform a particular task in our entire web application.
Eg: loan app
   registration app
   polling app etc

Diagram

Project = Several Applications + Configuration Information

Note:

1) The Django applications can be plugged into other projects.ie these are reusable. (Pluggable Django Applications)
2) Without existing Django project there is no chance of existing Django Application. Before creating any application first we required to create project.

How to create Django Project:

Once we installed django in our system, we will get 'django-admin' command line tool, which can be used to create our Django project.

```
django-admin startproject firstProject

D:\>mkdir djangoprojects

D:\>cd djangoprojects

D:\djangoprojects>django-admin start-project firstProject
```

The following project structure will be created

```
D:\djangoprojects>
 |
 +---firstProject
    ¦
    ¦---manage.py
    ¦
   +---firstProject
       ¦---settings.py
       ¦---urls.py
       ¦--wsgi.py
       ¦-- __init__.py
```

__init__.py:

It is a blank python script.Because of this special file name, Django treated this folder as python package.

Note: If any folder contains __init__.py file then only that folder is treated as Python package.But this rule is applicable until Python 3.3 Version.

settings.py:

In this file we have to specify all our project settings and and configurations like

installed applications, middileware configurations, database configurations etc

urls.py:

Here we have to store all our url-patterns of our project.

For every view (web page), we have to define separate url-pattern. End user can use url-patterns to access our webpages.

wsgi.py:

wsgi $\rightarrow$ Web Server Gateway Interface.
We can use this file while deploying our application in production on online server.

manage.py:

The most commonly used python script is manage.py

It is a command line utility to interact with Django project in various ways like to run development server, run tests, create migrations etc.

How to run Django Development server:

We have to move to the manage.py file location and we have to execute the following command.

```
py manage.py runserver

D:\djangoprojects\firstProject>py manage.py startserver

Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until you apply the
migrations for app(s): admin, auth, contenttypes, sessions. Run 'python manage.py migrate'
to apply them.
August 03, 2018 - 15:38:59
Django version 1.11, using settings 'firstProject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Now the server started.

How to send first request:

Open browser and send request:

http://127.0.0.1:8000/

The following should be response if everything goes fine.

It worked!

Congratulations on your first Django-powered page.
Next, start your first app by running python manage.py startapp [app_label].

You're seeing this message because you have DEBUG = True in your Django settings file and you haven't configured any URLs. Get to work!

Role of Web Server:

Web Server provides environment to run our web applications.

Web Server is responsible to receive the request and forward request to the corresponding web component based on url-pattern and to provide response to the end user.

Django framework is responsible to provide development server. Even Django framework provides one inbuilt database sqlite. Special Thanks to Django.

Note: Once we started Server a special database related file will be generated in our project folder structure.

db.sqlite3

Creation of First web application:

Once we creates Django project, we can create any number of applications in that project.

The following is the command to create application.

```
python manage.py startapp firstApp
```

```
D:\djangoprojects\firstProject>python manage.py startapp firstApp
```

The following is the folder structure got created.

```
D:\djangoprojects>
└───firstProject
    │ db.sqlite3
    │ manage.py
 │
    ├───firstApp
    │  │ admin.py
    │  │ apps.py
    │  │ models.py
    │  │ tests.py
    │  │ views.py
    │  │ __init__.py
    │  │
    │  └───migrations
    │        __init__.py
```

```
   |
   └──firstProject
       | settings.py
       | urls.py
       | wsgi.py
       | __init__.py
 |
```

Note: Observe that Application contains 6 files and project contains 4 files+ one special file: manage.py

1) __init__.py:
It is a blank Python script. Because of this special name,Python treated this folder as a package.

2) admin.py:
We can register our models in this file. Django will use these models with Django's admin interface.

3) apps.py:
In this file we have to specify application's specific configurations.

4) models.py:
In this file we have to store application's data models.

5) tests.py:
In this file we have to specify test functions to test our code.

6) views.py:
In this file we have to save functions that handles requests and return required responses.

7) migrations folder:
This directory stores database specific information related to models.

Note: The most important commonly used files in every project are views.py and models.py

Activities required for Application:

Activity-1: Add our application in settings.py,so that Django aware about our application.

In settings.py:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'firstApp'
]
```

Activity-2: Create a view for our application in views.py.
View is responsible to prepare required response to the end user. i.e view contains business logic.
There are 2 types of views.

1. Function Based Views
2. Class Based Views

In this application we are using Function based views.

views.py:

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def display(request):
    s='<h1>Hello Students welcome to DURGASOFT Django classes!!!</h1>'
    return HttpResponse(s)
```

Note:
1. Each view will be specified as one function in views.py.
In the above example display is the name of function which is nothing but one view.
2. Each view should take atleast one argument (request)
3. Each view should return HttpResponse object with our required response.

Diagram

View can accept request as input and perform required operations and provide proper response to the end user.

Activity-3: Define url-pattern for our view in urls.py file.
This url-pattern will be used by end-user to send request for our views.
The 'urlpatterns' list routes URLs to views.

For functional views we have to do the following 2 activities:
1. Add an import: from firstApp import views
2. Add a URL to urlpatterns:
   url(r'^greeting/', views.display)

urls.py:

```python
from django.conf.urls import url
from django.contrib import admin
from firstApp import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^greetings/', views.display),
]
```

Whenever end user sending the request with urlpattern: greeting then disply() function will be executed and provide required response.

Activity-4: Start Server and Send the request

py manage.py runserver

http://127.0.0.1:8000/greetings

Http Request flow in Django Application:

Diagram

1. Whenever end user sending the request first Django development server will get that request.

2. From the Request django will identify urlpattern and by using urls.py, the corresponding view will be identified.

3. The request will be forwarded to the view. The corresponding function will be executed and provide required response to the end user.

Summary of sequence of activities related to Django Project:

1) Creation of Django project
   django-admin startproject firstProject

2) Creation of Application in that project
   py manage.py startapp firstApp

3) Add application to the Project
   (inside settings.py)

4) Define view function inside views.py
5) Define url-pattern for our view inside urls.py

6) Start Server
   py manage.py runserver

7) Send the request

How to change Django Server Port:

By default Django develoment server will run on port number: 8000. But we can change port number based on our requirement as follows.

```
py manage.py runserver 7777
```

Now Server running on port number: 7777
We have to send the request with this port number only

http://127.0.0.1:7777/greetings/
http://127.0.0.1:8000/time/

Defining urlpatterns at application level instead of project level:

A Django project can contain multiple applications and each application can contain multiple views.Defining url-patterns for all views of all applications inside urls.py file of project creates maintenance problems and reduces reusability of applications.

We can solve this problem by defining url-patterns at application level instead of project level. For every application we have to create a separate urls.py file and we have to define all that application specific urls in that file. We have to link this application level urls.py file to project level urls.py file by using include() method.

Demo Application:

1. Creation of Project
django-admin startproject urlProject

2. Creation of Application
py manage.py startapp urlApp

3. Add our application to the Project inside settings.py file

```
INSTALLED_APPS=[
    .........
        'urlApp'
        ]
```

4. Define View Function in views.py

```
from django.http import HttpResponse
def appurlinfo(request):

  s='<h1>Application Level urls Demo</h1>'
  return HttpResponse(s)
```

5. Create a seperate urls.py file inside application

```
from django.conf.url import url
 from urlApp import views
 urlpatterns=[
    url(r'^test/',views.appurlinfo)
    ]
```

6. Include this application level urls.py inside project level urls.py file.
from django.conf.urls. import include

```
urlpatterns=[
  ....
  url(r'^urlApp/',include('urlApp.urls'),
  ]
```

6. Run Server
py manage.py runserver

7. Send Request
http://127.0.0.1:8000/urlApp/test

Advantages:

The main advantages of defining urlpatterns at application level instead of project level are

1. It promotes reusability of Django Applications across multiple projects
2. Project level urls.py file will be clean and more readable

Django Templates:

It is not recommended to write html code inside python script (views.py file) because:

1. It reduces readability because Python code mixed with html code

2. No seperation of roles. Python developer has to concentrate on both python code and html code.

3. It does not promote reusability of code

We can overcome these problems by seperating html code into a seperate html file.This html file is nothing but template.
From the Python file (views.py file) we can use these templates based on our requirement.

We have to write templates at project level only once and we can use these in multiple applications.

Python stuff required to develop Template Based Application:

1. To know the current Python file name

```
print(__file__) #test.py
```

2. To know absolute path of current Python File Name

```
import os
print(os.path.abspath(__file__))
```

Output: D:\durgaclasses\test.py

3. To know Base Directory name of the current file

```
print(os.path.dirname(os.path.abspath(__file__)))
```

Output: D:\durgaclasses

4. Inside D:\durgaclasses there is one folder named with templates. To know its absolute path

```
import os
BASE_DIR=os.path.dirname(os.path.abspath(__file__))
TEMPLATE_DIR=os.path.join(BASE_DIR,'templates')
print(TEMPLATE_DIR)
```

Output: D:\durgaclasses\templates

Note: The main advantage of this approach is we are not required to hard code system specific paths(locations) in our python script.

Steps to develop Template Based Application:

1. Creation of Project
   django-admin startproject templateProject

2. Creation of Application
   py manage.py startapp testApp

3. Add this application to the project in settings.py file,so that Django aware of application

4. Create a 'templates' folder inside main project folder.
   In that templates folder create a seperate folder named with testApp to hold that particular application specific templates.

5. Add templates folder to settings.py file so that Django can aware of our templates.

```
TEMPLATES = [
  {
    ...,
    'DIRS': ['D:\djangoprojects\templateProject\templates'],

  },]
```

It is not recommended to hard code system specific locations in settings.py file. To overcome this problem, we can generate templates directory path programatically as follows.

```
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATE_DIR=os.path.join(BASE_DIR,'templates')
```

Specify this TEMPLATE_DIR inside settings.py as follows

```
TEMPLATES = [
  {
    ...,
    'DIRS': [TEMPLATE_DIR,],

  },]
```

6. Create html file inside templateProject/templates/testApp folder. This html file is nothing but template.

<u>wish.html:</u>

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>First Template Page</title>
  </head>
```

```
  <body>
    <h1>Hello welcome to Second Hero of MVT: Templates</h1>
  </body>
</html>
```

7. Define Function based view inside views.py file

```
from django.shortcuts import render
def wish(request):
    return render(request,'testApp/wish.html')
```

8. Define URL Pattern either at application level or at project level

9. Run server and send request

Template Tags:

From Python views.py we can inject dynamic content to the template file by using template tags.

Template Tags also known as Template Variables.

Take special care about Template tag syntax it is not python syntax and not html syntax. Just it is special syntax.

Template tag syntax for inserting text data:
{ {insert_date} }

This template tag we have to place inside template file (ie html file) and we have to provide insert_date value from python views.py file.

Diagram

Demo Application to send date and time from views.py to template file:

wish.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>First Template Page</title>
```

```
    <style >
    h1{
      color:white;
      background: red;
    }
    </style>
  </head>
  <body>
    <h1>Hello Server Current Date and Time : <br>
      {{insert_date}}
    </h1>
  </body>
</html>
```

views.py:

```
from django.shortcuts import render
import datetime
def wish(request):
    date=datetime.datetime.now()
    my_dict={'insert_date':date}
    return render(request,'testApp/wish.html',context=my_dict)
```

Note: The values to the template variables should be passed from the view in the form of dictionary as argument to context.

Application to wish end user based on time:

wish.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>First Template Page</title>
    <style >
    #h11{
      color:red;
    }
    #h12{
      color:green;
    }
    </style>
  </head>
  <body>
    <h1 id=h11>{{insert_msg}}</h1>
    <h1 id=h12>Current Date and Time : {{insert_date}}</h1>
  </body>
```

```
</html>
```

views.py:

```
from django.shortcuts import render
import datetime

# Create your views here.
def wish(request):
    date=datetime.datetime.now()
    msg='Hello Guest !!!! Very Very Good '
    h=int(date.strftime('%H'))
    if h<12:
        msg +='Morning!!!'
    elif h<16:
        msg +='AfterNoon!!!'
    elif h<21:
        msg +='Evening!!!'
    else:
        msg='Hello Guest !!!! Very Very Good Night!!!'
    my_dict={'insert_date':date,'insert_msg':msg}
    return render(request,'testApp/wish.html',context=my_dict)
```

Working with Static Files:

Up to this just we injected normal text data into template by using template tags.

But sometimes our requirement is to insert static files like images,css files etc inside template file.

Process to include static files inside template:

1. Create a folder named with 'static' inside main project folder. It is exactly same as creating 'templates' folder.
In that 'static' folder create 'images' folder to place image files.

2. Add static directory path to settings.py file, so that Django can aware of our images.

settings.py:

```
STATIC_DIR=os.path.join(BASE_DIR,'static')

..
STATIC_URL = '/static/'

STATICFILES_DIRS=[
STATIC_DIR,
]
```

3. Make sure all paths are correct or not

http://127.0.0.1:8000/static/images/divine3.jpg

4. Use Template Tags to insert image
At the beginning of html just after <!DOCTYPE html> we have to include the following template tag

{% load staticfiles %}

Just we are conveying to the Django to load all static files.

We have to include image file as follows
<img src="{% static "images/divine3.jpg"% }">

wish.html:

```
<!DOCTYPE html>
{% load staticfiles %}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>First Template Page</title>
    <style >
    #h11{
      color:red;
    }
    #h12{
      color:green;
    }
    </style>
  </head>
  <body>
    <h1 id=h11>{{insert_msg}}</h1>
    <h1 id=h12>Current Date and Time : {{insert_date}}</h1>
    <h1>This climate preferable image is:</h1>
    <img src="{% static "images/divine3.jpg" %}" alt="">
  </body>
</html>
```

views.py:

```
from django.shortcuts import render
import datetime

# Create your views here.
def wish(request):
    date=datetime.datetime.now()
    msg=None
    h=int(date.strftime('%H'))
    if h<12:
        msg='Hello Guest !!!! Very Very Good Morning!!!'
    elif h<16:
        msg='Hello Guest !!!! Very Very Good AfterNoon!!!'
    elif h<21:
        msg='Hello Guest !!!! Very Very Good Evening!!!'
    else:
        msg='Hello Guest !!!! Very Very Good Night!!!'
    my_dict={'insert_date':date,'insert_msg':msg}
    return render(request,'testApp/wish.html',context=my_dict)
```

How to include css file:

1. Create a folder 'css' inside static and place our demo.css file in that css folder.

demo.css:

```
img{
  height: 500px;
  width: 500px;
  border: 10px red groove;
  margin:0% 20%;
}
h1{
  color:blue;
  text-align: center;
}
```

2. In the template html file we have to include this css file. We have to do this by using link tag inside head tag.

<link rel="stylesheet" href="{% static "css/demo.css" %}" >

DURGASOFT NEWS PORTAL:

index.html:

```
<!DOCTYPE html>
{% load staticfiles%}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" href="{% static "css/demo.css"%}">
  </head>
  <body>
    <h1>Welcome to DURGASOFT NEWS PORTAL</h1>
    <ul>
      <li> <a href="/movies">Movies Information</a> </li>
      <li> <a href="/sports">Sports Information</a> </li>
      <li> <a href="/politics">Politics Information</a> </li>
    </ul>
  </body>
</html>
```

news.html:

```
<!DOCTYPE html>
{% load staticfiles %}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" href="{% static "css/demo.css"%}">
  </head>
  <body>
      <h1>{{head_msg}}</h1>
      <ul>
        <li> <h2>{{sub_msg1}}</h2> </li>
        <li> <h2>{{sub_msg2}}</h2> </li>
        <li> <h2>{{sub_msg3}}</h2> </li>
      </ul>
      <img src="{% static "images/sunny.jpg" %}" alt="">
      <img src="{% static "images/guido.jpg" %}" alt="">
      <img src="{% static "images/divine3.jpg" %}" alt="">
  </body>
</html>
```

views.py:

```
from django.shortcuts import render

# Create your views here.
```

```
def moviesInfo(request):
    my_dict={'head_msg':'Movies Information',
             'sub_msg1':'Sonali slowly getting cured',
             'sub_msg2':'Bahubali-3 is just planning',
             'sub_msg3':'Salman Khan ready to marriage',
             'photo':'images/sunny.jpg'}
    return render(request,'news.html',context=my_dict)
def sportsInfo(request):
    my_dict={'head_msg':'Sports Information',
             'sub_msg1':'Anushka Sharma Firing Like anything',
             'sub_msg2':'Kohli updating in game anything can happend',
             'sub_msg3':'Worst Performance by India-Sehwag',
             }
    return render(request,'news.html',context=my_dict)
def politicsInfo(request):
    my_dict={'head_msg':'Politics Information',
             'sub_msg1':'Achhce Din Aaa gaya',
             'sub_msg2':'Rupee Value now 1$:70Rs',
             'sub_msg3':'In India Single Paisa Black money No more',
             }
    return render(request,'news.html',context=my_dict)
def index(request):
    return render(request,'index.html')
```

<u>settings.py:</u>

```
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATES_DIR=os.path.join(BASE_DIR,'templates')
STATIC_DIR=os.path.join(BASE_DIR,'static')

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'newsApp'
]

TEMPLATES = [
    {
        '
        'DIRS': [TEMPLATES_DIR,],

            ],
        },
```

```
    },
]

STATIC_URL = '/static/'
STATICFILES_DIRS=[
STATIC_DIR,
]
```

urls.py:

```
from django.conf.urls import url
from django.contrib import admin
from newsApp import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^movies/', views.moviesInfo),
    url(r'^sports/', views.sportsInfo),
    url(r'^politics/', views.politicsInfo),
    url(r'^$', views.index),
]
```

Single Project with Multiple applications:

urls.py:

```
from django.conf.urls import url
from django.contrib import admin
from firstApp import views as v1
from secondApp import views as v2

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^firstwish/', v1.wish1),
    url(r'^secondwish/', v2.wish2),
]
```

Working with Models and Databases:

As the part of web application development, compulsory we required to interact with database to store our data and to retrieve our stored data.

Django provides a big in-built support for database operations. Django provides one inbuilt database sqlite3.
For small to medium applications this database is more enough. Django can provide support for other databases also like oracle, mysql,postgresql etc

Database configuration:

Django by default provides sqlite3 database. If we want to use this database,we are not required to do any configurations.

The default sqllite3 configurations in settings.py file are declared as follows.

settings.py:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

If we don't want sqlite3 database then we have to configure our own database with the following parameters.

1. ENGINE: Name of Database engine
2. NAME: Database Name
3. USER: Database Login user name
4. PASSWORD: Database Login password
5. HOST: The Machine on which database server is running
6. PORT: The port number on which database server is running

Note: Most of the times HOST and PORT are optional.

How to check django database connection:

We can check whether django database configurations are properly configured or not by using the following commands from the shell

```
D:\djangoprojects\modelProject>py manage.py shell
>    from django.db import connection
>    c=connection.cursor()
```

If we are not getting any error means our database configurations are proper.

Configuration of mysql database:

First we have to create our own logical database in the mysql.

```
mysql> create database employeedb;
```

We have to install mysqlclient by using pip as follows

pip install --only-binary :all: mysqlclient

<u>settings.py:</u>

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.mysql',
    'NAME': 'employeedb',
    'USER':'root',
    'PASSWORD':'root'
  }
}
```

Checking configurations:

```
D:\djangoprojects\modelProject>py manage.py shell
>   from django.db import connection
>   c=connection.cursor()
```

Configuration of Oracle database:

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.oracle',
    'NAME': 'XE',
    'USER':'scott',
    'PASSWORD':'tiger'
  }
}
```

Note: We can find oracle database name by using the following command.

```
SQL> select * from global_name;
```

Model Class:

- A Model is a Python class which contains database information.
- A Model is a single, definitive source of information about our data. It contains fields and behavior of the data what we are storing.
- Each model maps to one database table.
- Every model is a Python class which is the child class of (django.db.models.Model)
- Each attribute of the model represents a database field.
- We have to write all model classes inside 'models.py' file.

1. Create a project and application and link them.

```
django-admin startproject modelProject
cd modelProject/
python manage.py startapp testApp
```

After creating a project and application, in the models.py file, write the following code:

models.py:

```python
from django.db import models

# Create your models here.

class Employee(models.Model):
    eno=models.IntegerField()
    ename=models.CharField(max_length=30)
    esal=models.FloatField()
    eaddr=models.CharField(max_length=30)
```

Note: This model class will be converted into Database table. Django is responsible for this.

table_name: appName_Employee
fields: eno, ename, esal and eaddr. And one extra field: id
behaviors: eno is of type Integer, ename is of type Char and max_length is 30 characters.

Hence,

Model Class = Database Table Name + Field Names + Field Behaviors

Converting Model Class into Database specific SQL Code:

Once we write Model class, we have to generate the corresponding SQL Code. For this, we have to use "makemigrations" command.

```
 Python manage.py make migrations
```

It results the following :

```
Migrations for 'testApp':
  testApp/migrations/0001_initial.py
- Create model Employee
```

How to see corresponding sql code of migrations:

To see the generated SQL Code, we have to use the following command "sqlmigrate"

python manage.py sqlmigrate testApp 0001

```
BEGIN;
--
-- Create model Employee
--
CREATE TABLE "testApp_employee" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "eno" integer NOT NULL, "ename" varchar(30) NOT NULL, "esal" real NOT NULL, "eaddr" varchar(30) NOT NULL);
COMMIT;
```

Note: Here 0001 is the file passed as an argument

"id" field:
1. For every table(model), Django will generate a special column named with "id".
2. ID is a Primary Key. (Unique Identifier for every row inside table is considered as a primary key).
3. This field(id) is auto increment field and hence while inserting data, we are not required to provide data for this field.
4. This id field is of type "AutoField"
5. We can override the behavior of "id" field and we can make our own field as "id".
6. Every Field is by default "NOT NULL".

How to execute generated SQL code (migrate command):

After generating sql code, we have to execute that sql code to create table in database. For this, we have to use 'migrate' command.

python manage.py migrate

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, testApp
Running migrations:
 Applying contenttypes.0001_initial... OK
 Applying auth.0001_initial... OK
 Applying admin.0001_initial... OK
 Applying admin.0002_logentry_remove_auto_add... OK
 Applying contenttypes.0002_remove_content_type_name... OK
 Applying auth.0002_alter_permission_name_max_length... OK
 Applying auth.0003_alter_user_email_max_length... OK
 Applying auth.0004_alter_user_username_opts... OK
 Applying auth.0005_alter_user_last_login_null... OK
 Applying auth.0006_require_contenttypes_0002... OK
 Applying auth.0007_alter_validators_add_error_messages... OK
 Applying auth.0008_alter_user_username_max_length... OK
 Applying sessions.0001_initial... OK
 Applying testApp.0001_initial... OK
```

Note: Now tables will be created in the database.

What is the advantage of creating tables with ‘migrate’ Command

If we use ‘migrate’ command, then all Django required tables will be created in addition to our application specific tables. If we create table manually with sql code, then only our application specific table will be created and django my not work properly. Hence it is highly recommended to create tables with ‘migrate’ command.

How to check created table in django admin interface:

We have to register model class in ‘admin.py’ file.

admin.py:

```python
from django.contrib import admin
from testApp.models import Employee

# Register your models here.

admin.site.register(Employee)
```

Creation of super user to login to admin interface:

We can create super user with the following command by providing username, mailid, password.

```
python manage.py createsuperuser
```

We can login to admin interface → Start the server and login to admin interface using the created credentials.

```
python manage.py runserver
```

Open the following in browser:
http://127.0.0.1:8000/admin/

Difference between makemigrations and migrate:

“makemigrations” is responsible to generate SQL code for Python model class whereas “migrate” is responsible to execute that SQL code so that tables will be created in the database.

To display data in admin interface in browser:

models.py:

```python
from django.db import models

# Create your models here.
```

```python

class Employee(models.Model):
    eno=models.IntegerField()
    ename=models.CharField(max_length=30)
    esal=models.FloatField()
    eaddr=models.CharField(max_length=30)

    def __str__(self):
        return 'Employee Object with eno: +str(self.no)'
```

admin.py:

```python
from django.contrib import admin
from testApp.models import Employee

# Register your models here.

class EmployeeAdmin(admin.ModelAdmin):
    list_display=['eno','ename','esal','eaddr']

admin.site.register(Employee,EmployeeAdmin)
```

Now we can write views to get data from the database and send to template.

Before writing views.py file, create “templates” and “static” folder with respective application folders and HTML and CSS files and link them in settings.py file.

Views.py:

```python
from django.shortcuts import render
from testApp.models import Employee

# Create your views here.
def empdata(request):
    emp_list=Employee.objects.all()
    my_dict={'emp_list':emp_list}
    return render(request, 'testApp/emp.html', context=my_dict)
```

emp.html

```html
<!DOCTYPE html>
{% load staticfiles %}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" href="{% static '/css/demo.css'%}">
  </head>

  <body>
```

```
<h1> The employees list is : </h1>

{% if emp_list %}
<table>
  <thead>
    <th> eno </th>
    <th> ename </th>
    <th> esal </th>
    <th> eaddr </th>
  </thead>

  {% for emp in emp_list %}
  <tr>
    <td> {{emp.eno}}</td>
    <td>{{emp.ename}}</td>
    <td>{{emp.esal}}</td>
    <td> {{emp.eaddr}}</td>
  </tr>
  {% endfor %}

</table>
{%else%}
<p> No records found </p>
{% endif %}

</body>
</html>
```

MVT Diagram:

FAQs:

How to configure database inside settings.py?
How to check connections?
How to define Model class inside models.py
How we can perform makemigrations?
How we can perform migrate?
How to add our model to admin interface inside admin.py
To display total data how to write ModelAdmin class inside admin.py
how to createsuperuser?
How to login to admin interface and add data to our tables?
How to see generated sqlcode b'z of makemigrations

Faker Module:

We can use Faker Module to generate fake data for our database models.

```
from faker import Faker
from random import *
fakegen=Faker()
name=fakegen.name()
print(name)
first_name=fakegen.first_name()
last_name=fakegen.last_name()
print(first_name)
print(last_name)
date=fakegen.date()
print(date)
number=fakegen.random_number(5)
print(number)
email=fakegen.email()
print(email)
print(fakegen.city())
print(fakegen.random_int(min=0, max=9999))
print(fakegen.random_element(elements=('Project Manager', 'TeamLead', 'Software Engineer')))
```

<u>Note:</u> Working with mysql db(studentinfo project)

Django Forms:

It is the very important concept in web development.
The main purpose of forms is to take user input.

<u>Eg:</u> login form, registration form, enquiry form etc

From the forms we can read end user provided input data and we can use that data based on requirement. We may store in the database for future purpose. We may use just for validation/authentication purpose etc

Here we have to use Django specific forms but not HTML forms.

Advantages of Django Forms over HTML forms:

1. We can develop forms very easily with python code
2. We can generate HTML Form widgets/components (like textarea, email, pwd etc) very quickly
3. Validating data will become very easy
4. Processing data into python data structures like list, set etc will become easy
5. Creation of Models based on forms will become easy etc.

Process to generate Django forms:

Step-1: Creation of forms.py file in our application folder with our required fields.

forms.py:

```
from django import forms
class StudentForm(forms.Form):
    name=forms.CharField()
    marks=forms.IntegerField()
```

Note: name and marks are the field names which will be available in html form

Step-2: usage of forms.py inside views.py file:

views.py file is responsible to send this form to the template html file

views.py:

```
from django.shortcuts import render
from . import forms

# Create your views here.
def studentinputview(request):
    form=forms.StudentForm()
    my_dict={'form':form}
    return render(request,'testapp/input.html',context=my_dict)
```

# **alternative short way:**

```
def studentinputview(request):
    form=forms.StudentForm()
    return render(request,'testapp/input.html',{'form':form})
```

Note: context parameter is optional.We can pass context parameter value directly without using keyword name 'context'

Step-3: Creation of html file to hold form:

Inside template file we have to use template tag to inject form

{{form}}

It will add only form fields. But there is no <form> tag and no submit button.
Even the fields are not arranged properly.It is ugly form.

We can make proper form as follows

```
<h1>Registration Form</h1>
 <div class="container" align="center">
   <form  method="post">
  {{form.as_p}}
  <input type="submit" class="btn btn-primary" name="" value="Submit">
   </form>

 </div>
```

<u>input.html:</u>

```
<!DOCTYPE html>
{%load staticfiles%}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="{%static "css/bootstrap.css"%}">
    <link rel="stylesheet" href="{%static "css/demo2.css"%}">
    <title></title>
  </head>
  <body>
    <h1>Registration Form</h1>
    <div class="container" align="center">
      <form  method="post">
          {{form.as_p}}
          <input type="submit" class="btn btn-primary" name="" value="Submit">
      </form>
    </div>
  </body>
</html>
```

If we submit this form we will get 403 status code response
Forbidden (403)
CSRF verification failed. Request aborted.

Help
Reason given for failure:

   CSRF token missing or incorrect.

Every form should satisfy CSRF (Cross Site Request Forgery) Verification, otherwise Django won't accept our form.
It is meant for website security. Being a programmer we are not required to worry anything about this. Django will takes care everything.
But we have to add csrf_token in our form.

```
<h1>Registration Form</h1>
   <div class="container" align="center">
        <form  method="post">
```

```
        {{form.as_p}}
        {% csrf_token %}
        <input type="submit" class="btn btn-primary" name="" value="Submit">
      </form>
  </div>
```

If we add csrf_token then in the generate form the following hidded field will be added,which makes our post request secure

<input type='hidden' name='csrfmiddlewaretoken' value='1ZqIJJqTLMVa6RFAyPJh7pwzyFmdiHzytLxJIDzAkKULJz4qHcetLoKEsRLwyz4h'/>

The value of this hidden field is keep on changing from request to request.Hence it is impossible to forgery of our request.

If we configured csrf_token in html form then only django will accept our form.

How to process input data from the form inside views.py file:

We required to modify views.py file. The end user provided input is available in a dictionary named with 'cleaned_data'

views.py:

```
from django.shortcuts import render
from . import forms

# Create your views here.
def studentinputview(request):
    form=forms.StudentForm()
    if request.method=='POST':
        form=forms.StudentForm(request.POST)
        if form.is_valid():
            print('Form validation success and printing data')
            print('Name:',form.cleaned_data['name'])
            print('Marks:',form.cleaned_data['marks'])
    return render(request,'testapp/input.html',{'form':form})
```

project: formproject

Student FeedBack Form Project

forms.py:

```python
from django import forms

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)
```

views.py:

```python
from django.shortcuts import render
from . import forms

def feedbackview(request):
    form=forms.FeedBackForm()
    if request.method=='POST':
        form=forms.FeedBackForm(request.POST)
        if form.is_valid():
            print('Form Validation Success and printing information')
            print('Name:',form.cleaned_data['name'])
            print('Roll No:',form.cleaned_data['rollno'])
            print('Email:',form.cleaned_data['email'])
            print('FeedBack:',form.cleaned_data['feedback'])
    return render(request,'testapp/feedback.html',{'form':form})
```

feedback.html:

```html
<!DOCTYPE html>
{% load staticfiles%}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
    <link rel="stylesheet" href="{% static "css/demo4.css" %}">
    <title></title>
  </head>
  <body>
    <div class="container" align='center'>
      <h1>Student Feedback Form</h1><hr>
      <form class="" action="index.html" method="post">
        {{form.as_p}}
        {% csrf_token %}
        <input type="submit" class="btn btn-primary" value="Submit Feedback">
      </form>
    </div>
  </body>
</html>
```

Form Validations:

Once we submit the form we have to perform validations like
1. Length of the field should not be empty
2. The max number of characters should be 10
3. The first character of the name should be 'd' etc

We can implement validation logic by using the following 2 ways.

1. Explicitly by the programmer by using clean methods
2. By using Django inbuilt validators

Note: All validations should be implemented in the forms.py file

1. Explicitly by the programmer by using clean methods:

The syntax of clean method:

clean_fieldname(self)

In the FormClass for any field if we define clean method then at the time of submit the form, Django will call this method automatically to perform validations. If the clean method won't raise any error then only form will be submitted.

```
from django import forms

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)

    def clean_name(self):
        inputname=self.cleaned_data['name']
        if len(inputname) < 4:
```

```
            raise forms.ValidationError('The Minimum no of characters in the name field should be 4')
        return inputname
```

The returned value of clean method will be considered by Django at the time of submitting the form.

forms.py:

```
from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)

    def clean_name(self):
        print('validating name')
        inputname=self.cleaned_data['name']
        if len(inputname) < 4:
            raise forms.ValidationError('The Minimum no of characters in the name field should be 4')
        return inputname+'durga'
    def clean_rollno(self):
        inputrollno=self.cleaned_data['rollno']
        print('Validating rollno field')
        return inputrollno
    def clean_email(self):
        inputemail=self.cleaned_data['email']
        print('Validating email field')
        return inputemail

    def clean_feedback(self):
        inputfeedback=self.cleaned_data['feedback']
        print('Validating feedback field')
        return inputfeedback
```

server console:

```
validating name
Validating rollno field
Validating email field
Validating feedback field
Form Validation Success and printing information
Name: Durgadurga
Roll No: 101
Email: durgaadvjava@gmail.com
FeedBack: This is sample feedback
```

Note:

1 .Django will call these filed level clean methods automatically and we are not required to call explicitly.

2. Form validation by using clean methods is not recommended.

2. Django's inbuilt core validators:

Django provides several inbuilt core validators to perform very common validations. We can use these validators directly and we are not required to implement.

Django's inbuilt validators are available in the django.core module.

from django.core import validators

To validate Max number of characters in the feedback as 40,we have to use inbuilt validators as follows.

forms.py:

```python
from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea,validators=[validators.MaxLengthValidator(40)])
```

Note: We can use any number of validators for the same field

feedback=forms.CharField(widget=forms.Textarea,validators=[validators.MaxLengthValidator(40), validators.MinLengthValidator(10)])

Note: Usage of built in validators is very easy when compared with clean methods.

How to implement custom validators by using the same validator parameter:

The value of name parameter should starts with 'd' or 'D'. We can implement this validation as follows

```python
def starts_with_d(value):
    if value[0].lower() != 'd':
        raise forms.ValidationError('Name should be starts with d | D')

class FeedBackForm(forms.Form):
    name=forms.CharField(validators=[starts_with_d])
    rollno=forms.IntegerField()
```

Validation of total form directly by using a single clean method:

Whenever we are submitting the form Django will call clean() method present in our Form class. In that method we can implement all validations.

forms.py:

```
from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)

    def clean(self):
        print('Total Form Validation...')
        total_cleaned_data=super().clean()
        inputname=total_cleaned_data['name']
        if inputname[0].lower() != 'd':
            raise forms.ValidationError('Name parameter should starts with d')
        inputrollno=total_cleaned_data['rollno']
        if inputrollno <=0:
            raise forms.ValidationError('Rollno should be > 0')
```

How to check original pwd and reentered pwd are same or not:

forms.py:

```
from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name=forms.CharField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='Re Enter Password',widget=forms.PasswordInput)
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)

    def clean(self):
        print('validating passwords match...')
        total_cleaned_data=super().clean()
        fpwd=total_cleaned_data['password']
```

```
        spwd=total_cleaned_data['rpassword']
        if fpwd != spwd:
            raise forms.ValidationError('Both passwords must be matched')
```

How to prevent requests from BOT:

Generally form requests can be send by end user. Sometimes we can write automated programming script which is responsible to fill the form and submit. This automated programming script is nothing but BOT.

The main objectives of BOT requests are:

1. To create unnecessary heavy traffic to the website, which may crash our application

2. To spread malware (viruses)

Being web developer compulsory we have to think about BOT requests and we have to prevent these requests.

How to prevent BOT Requests:

In the form we will place one hidden form field, which is not visible to the end user. Hence there is no chance of providing value to this hidden field.

But BOT will send the value for this hidden field also.If hidden field got some value means it is the request from BOT and prevent that form submission.

```
from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name=forms.CharField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='Re Enter Password',widget=forms.PasswordInput)
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)
    bot_handler=forms.CharField(required=False,widget=forms.HiddenInput)

    def clean(self):
        print('validating passwords match...')
        total_cleaned_data=super().clean()
```

```
        fpwd=total_cleaned_data['password']
        spwd=total_cleaned_data['rpassword']
        if fpwd != spwd:
            raise forms.ValidationError('Both passwords must be matched')
        bot_handler_value=total_cleaned_data['bot_handler']
        if len(bot_handler_value)>0:
            raise forms.ValidationError('Request from BOT...cannot be submitted!!!')
```

Notes with bot field:

```
class FeedBackForm(forms.Form):
    normal fields..
    bot_handler=forms.CharField(required=False,widget=forms.HiddenInput)

    def clean(self):
        total_cleaned_data=super().clean()
        bot_handler_value=total_cleaned_data['bot_handler']
        if len(bot_handler_value)>0:
            raise forms.ValidationError('Request from BOT...cannot be submitted!!!')
```

Note: Other ways to prevent BOT requests
1. By using Captchas
2. By using image recognizers
   (like choose 4 images where car present)

Note:
Other way to create project
py -m django startproject modelformproject

Model Forms (Forms based on Model):

Sometimes we can create form based on Model, such type of forms are called model based forms or model forms.

The main advantage of model forms is we can grab end user input and we can save that input data very easily to the database.

Django provides inbuilt support to develop model based forms very easily.

How to develop Model based forms:

1. While develop FormClass instead of inheriting forms.Form class,we have to inherit forms.ModelForm class.

```
class RegisterForm(forms.ModelForm):
    ....
```

2. We have to write one nested class (Meta class) to specify Model information and required fields.

```
class RegisterForm(forms.ModelForm):
   # field declarations if we are performing any custom validations.If we are not defining any
custom validations then here we are not required to specify any field.

   class Meta:
     # we have to specify Model class name and requied fields
     model=Student
     fields='__all__'
```

Case-1: Instead of all fields if we want only selected fields,then we have to specify as follows

```
class Meta:
   model=Student
   fileds=('field1','field2','field3')
```

In the form only 3 fields will be considered.
If Model class contains huge number of fields and we required to consider very less number of fields in the form then we should use this approach.

Case-2:

Instead of all fields if we want to exclude certain fields,then we have to specify as follows

```
class Meta:
   model=Student
   exclude=['field1','field2']
```

In the form all fields will be considered except field1 and field2.

If the Model class contains huge number of fields and if we want to exclude very few fields then we have to use this approach.

Q. In Model based forms, how many ways are there to specify fields information

Ans: 3 ways
1. All fields
2. Include certain fields
3. Exclude certain fields

Note: The most commonly used approach is to include all fields.

How to save user's input data to database in Model based forms:

We have to use save() method.

```
def student_view(request):
    ...
    if request.method=='POST':
       form=RegisterForm(request.POST)
            if form.is_valid():
              form.save(commit=True)
            ..
```

Demo project-1 (modelformproject):

models.py:

```
from django.db import models

# Create your models here.
class Student(models.Model):
   name=models.CharField(max_length=30)
   marks=models.IntegerField()
```

forms.py:

```
from django import forms
from testapp.models import Student
class StudentForm(forms.ModelForm):
   #fields with validations
   class Meta:
      model=Student
      fields='__all__'
```

views.py:

```
from django.shortcuts import render
from . import forms

# Create your views here.
def student_view(request):
   form=forms.StudentForm
   if request.method=='POST':
      form=forms.StudentForm(request.POST)
      if form.is_valid():
         form.save(commit=True)
```

```
    return render(request,'testapp/studentform.html',{'form':form})
```

Demo Project-2: movieproject

Advanced Templates:

1. Template Inheritance
2. Template Filters
3. Template tags for relative urls

1. Template Inheritance:

If multiple template files have some common code,it is not recommended to write that common code in every template html file. It increases length of the code and reduces readability. It also increases development time.

We have to seperate that common code into a new template file,which is also known as base template. The remaining template files should required to extend base template so that the common code will be inherited automatically.

Inheriting common code from base template to remaining templates is nothing but template inheritance.

How to implement template inheritane:

base.html:

```
<!DOCTYPE html>
 html,css,bootstrap links
 <body>
   common code required for every child tempalte
   {% block child_block%}
     Anything outside of this block available to child tag.
     in child template the specific code should be in this block
   {%endblock%}
 </body>
 </html>
```

child.html:

```
<!DOCTYPE html>
{%extends 'testapp/base.html'%}
  {% block child_block %}
     child specific extra code
  {%endblock%}
```

Demo program: advtempproject

base.html:

```
<!DOCTYPE html>
{%load staticfiles%}
<html lang="en" dir="ltr">
 <head>
   <meta charset="utf-8">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
   <link rel="stylesheet" href="{%static "css/advtemp.css"%}">
   <title></title>
 </head>
 <body>
   <nav class="navbar navbar-default navbar-fixed-top navbar-inverse">
       <div class="container-fluid">

         <div class="navbar-header">
           <a class="navbar-brand" href="/">DURGA NEWS</a>
         </div>
          <ul class="nav navbar-nav">
<li class="active"><a href="/">Home <span class="sr-only">(current)</span></a></li>
            <li><a href="/movies">Movies</a></li>
            <li><a href="/sports">Sports</a></li>
            <li><a href="/politics">Politics</a></li>
          </ul>
       </div><!-- /.container-fluid -->
 </nav>
<div class="container">
 {%block body_block%}
   <!-- outside of this block everything available to child tempaltes -->
 {%endblock%}
</div>
</body>
</html>
```

index.html:

```
<!DOCTYPE html>
{%extends 'testapp/base.html'%}
 {% block body_block %}
    <br><br><br><br><br><br>
   <h1>Welcome to DURGA NEWS PORTAL</h1>
 {%endblock%}
```

sports.html:

```
<!DOCTYPE html>
{%extends 'testapp/base.html'%}
  {% block body_block %}
  <br><br><br>
    <h1>Sports Information</h1>
  {%endblock%}
```

Advantages of Template Inheritance:

1. What ever code available in base template is by default available to child templates and we are not required to write again.Hence it promotes Code Reusability (DRY)

2. It reduces length of the code and improves readability

3. It reduces development time

4. It provides unique and same look and feel for total web application.

Note: Based on our requirement we can extend any number of base templates.i.e Multiple Inheritance is applicable for templates.

Tempalte Filters:

In the template file, the injected data can be displayed by using template tags.

{{emp.eno}}

Before displaying to the end user if we want to perform some modification to the injected text, like cut some information or converting to title case etc,then we should go for Template filters.

Syntax of Template Filter:

{{value|filtername:"argument"}}

Filter may take or may not take arguments.i.e arguments are optional.

Eg:
<li>{{msg1|lower}}</li>
  msg1 will be displayed in lower case

<li>{{msg3|add:"Durga"}}</li>
  "Durga" will be added to msg3 and then display the result to the end user

{{ msg|title }}

{{ my_date|date:"Y-m-d" }}

Note: There are tons of built in filters are available.

https://docs.djangoproject.com/en/2.1/ref/templates/builtins/#ref-templates-builtins-filters

How to create our own filter:

Based on our requirement we can create our own filter.

Steps:

1. Create a folder 'templatetags' inside our application folder

2. Create a special file named with __init__.py inside templatetags folder, so that Django will consider this folder as a valid python package

3. Create a python file inside templatetags folder to define our own filters

cust_filters.py -->any name

cust_filters.py:

```
from django import template
register=template.Library()

def first_eight_upper(value):
    """This is my own filter'''
    result=value[:8].upper()
    return result

register.filter('f8upper',first_eight_upper)
```

Note: We can also register filter with the decorator as follows.

```
from django import template
register=template.Library()

@register.filter(name='f8upper')
def first_eight_upper(value):
    """This is my own filter'''
    result=value[:8].upper()
    return result
```

f8upper is the name of the filter which can be used inside template file.

4. Inside template file we have to load the filter file as follows(In the child template bunot in base template)

{%load cust_filters%}

5. We can invoke the filter as follows

{{msg|f8upper}}

<u>movies.html:</u>

```
<!DOCTYPE html>
{%extends 'testapp/base.html'%}
 {% block body_block %}
  <h1>Movies Information</h1><hr>
  {%load cust_filters%}
  <ul>
   <li>{{msg1|lower}}</li>
   <li>{{msg2|upper}}</li>
   <li>{{msg3|add:"--Durga"}}</li>
   <li>{{msg4|f8upper}}</li>
   <li>{{msg5}}</li>
 </ul>
 {%endblock%}
```

<u>Eg 2:</u> Custom Filter with argument

```
@register.filter(name='c_and_c')
def cut_and_concate(value,arg):
  result=value[:4]+str(arg)
  return result
```

<u>Note:</u> The main advantage of template filters is we can display the same data in different styles based on our requirement.

advtemp project

<u>Template Tags for URLs:</u>

<a href='testapp1/thankyou'>Thank You </a>

<a href="{% url 'thankyou' %}">Thank You </a>

<a href="{% url 'testapp.views.thankyou' %}">Thank You </a>

<a href="{% url 'testapp:thankyou' %}">Thank You </a>

<u>Session Management:</u>

Client and Server can communicate with some common language which is nothing but HTTP.

The basic limitation of HTTP is, it is stateless protocol. i.e it is unable to remember client information for future purpose across multiple requests. Every request to the server is treated as new request.

Hence some mechanism must be required at server side to remember client information across multiple requests.This mechanism is nothing but session management mechanism.

The following are various session management mechanisms.

1. Cookies
2. Session API
3. URL Rewriting
4. Hidden Form
Fields etc

Session Management By using Cookies:

Cookie is a very small amount of information created by Server and maintained by client.

Diagram

Whenever client sends a request to the server,if server wants to remember client information for the future purpose then server will create cookie object with the required information. Server will send that Cookie object to the client as the part of response. Client will save that cookie in its local machine and send to the server with every consecutive request. By accessing cookies from the request server can remember client information.

How to test our browser supports Cookies or not:

We have to use the following 3 methods on the request object.

1. set_test_cookie()
2. test_cookie_worked()
3. delete_test_cookie()

views.py:

```python
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def index(request):
    request.session.set_test_cookie()
    return HttpResponse('<h1>index Page</h1>')

def check_view(request):
    if request.session.test_cookie_worked():
        print('cookies are working properly')
        request.session.delete_test_cookie()
        return HttpResponse('<h1>Checking Page</h1>')
```

Note: Before executing this program compulsory we should perform migrate.

Session Management by using Cookies:

views.py:

```python
from django.shortcuts import render

# Create your views here.
def count_view(request):
    if 'count' in request.COOKIES:
        newcount=int(request.COOKIES['count'])+1
    else:
        newcount=1
    response=render(request,'testapp/count.html',{'count':newcount})
    response.set_cookie('count',newcount)
    return response
```

count.html:

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
```

```
    <meta charset="utf-8">
    <title></title>
    <style >
    span{
      font-size: 200px;
      font-weight: 900;
    }

    </style>
  </head>
  <body>
    <h1>Page Count is: <span> {{count}}<span></h1>
  </body>
</html>
```

Session Management by using Cookies Demo Program-3:

```
from django.shortcuts import render
from testapp.forms import LoginForm
import datetime

# Create your views here.
def home_view(request):
    form=LoginForm()
    return render(request,'testapp/home.html',{'form':form})

def date_time_view(request):
    # form=LoginForm(request.GET)
    name=request.GET['name']
    response=render(request,'testapp/datetime.html',{'name':name})
    response.set_cookie('name',name)
    return response

def result_view(request):
    name=request.COOKIES['name']
    date_time=datetime.datetime.now()
    my_dict={'name':name,'date_time':date_time}
    return render(request,'testapp/result.html',my_dict)
```

home.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
```

```
    <h1>Welcome to DURGASOFT</h1>
    <form action="/second">
     {{form.as_p}}
     {%csrf_token%}
     <input type="submit" name="" value="Enter Name">
    </form>
  </body>
</html>
```

<u>datetime.html:</u>

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}}</h1> <hr>
    <a href="/result">Click Here to get Date and Time</a>
  </body>
</html>
```

<u>result.html:</u>

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}}</h1><hr>
    <h1>Current Date and Time:{{date_time}}</h1>
    <a href="/result">Click Here to get Updated Date and Time</a>
  </body>
</html>
```

<u>Cookie Example-4:</u>

Diagram

views.py:

```python
from django.shortcuts import render

# Create your views here.
def name_view(request):
    return render(request,'testapp/name.html')

def age_view(request):
    name=request.GET['name']
    response=render(request,'testapp/age.html',{'name':name})
    response.set_cookie('name',name)
    return response

def gf_view(request):
    age=request.GET['age']
    name=request.COOKIES['name']
    response=render(request,'testapp/gf.html',{'name':name})
    response.set_cookie('age',age)
    return response

def results_view(request):
    name=request.COOKIES['name']
    age=request.COOKIES['age']
    gfname=request.GET['gfname']
response=render(request,'testapp/results.html',{'name':name,'age':age,'gfname':gfname})
    response.set_cookie('gfname',gfname)
    return response
```

name.html:

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Welcome to DURGASOFT</h1>
    <form action='/age'>
      Enter Name: <input type="text" name="name" value=""><br><br>
      <input type="submit" name="" value="Submit Name">
    </form>

  </body>
</html>
```

age.html:

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}}..</h1><hr>
      <form action='/gf'>
        Enter Age: <input type="text" name="age" value=""><br><br>
        <input type="submit" name="" value="Submit Age">
      </form>
  </body>
</html>
```

gf.html:

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}}..</h1><hr>

      <form action='/results'>
        Enter Girl Friend Name: <input type="text" name="gfname" value=""><br><br>
        <input type="submit" name="" value="Submit GFName">
      </form>

  </body>
</html>
```

results.html:

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}} Thanks for providing info</h1>
    <h2>Please cross check your data and confirm</h2><hr>
    <ul>
      <li>Name:{{name}}</li>
      <li>Age:{{age}}</li>
      <li>Girl Friend Name:{{gfname}}</li>
```

```
    </ul>
  </body>
</html>
```

Limitations of Cookies:

1. By using cookies we can store very less amount of information. The size of the cookie is fixed.Hence if we want to store huge amount of information then cookie is not best choice.

2. Cookie can hold only string information. If we want to store non-string objects then we should not use cookies.

3. Cookie information is stored at client side and hence there is no security

4. Everytime with every request, browser will send all cookies related to that application,which creates network traffic problems.

5. There is a limit on the max number of cookies supported by browser.

To overcome all these limitations we should go for sessions.

Temporary vs Permanent Cookies:

If we are not setting any max_age for the cookie,then the cookies will be stored in browsers cache.Once we closed browser automatically the cookies will be expired.Such type of cookies are called temporary Cookies.

We can set temporary Cookie as follows:
response.set_cookie(name,value)

If we are setting max_age for the cookie,then cookies will be stored in local file system permanently.Once the specified max_age expires then only cookies will be expired.Such type of cookies are called permanent or persistent cookies. We can set Permanent Cookies as follows

response.set_cookie(name,value,max_age=180)
response.set_cookie(name,value,180)

The time unit for max_age is in seconds.

Demo Program-3:

views.py:

```
from django.shortcuts import render
from testapp.forms import ItemAddForm

```

```python
# Create your views here.
def index(request):
    return render(request,'testapp/home.html')
def additem(request):
    form=ItemAddForm()
    response=render(request,'testapp/additem.html',{'form':form})
    if request.method=='POST':
        form=ItemAddForm(request.POST)
        if form.is_valid():
            name=form.cleaned_data['itemname']
            quantity=form.cleaned_data['quantity']
            response.set_cookie(name,quantity,180)
            # return index(request)
    return response
def displayitem_view(request):
    return render(request,'testapp/showitems.html')
```

<u>home.html:</u>

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
    <title></title>
  </head>
  <body>
    <div class="container" align='center'>
      <div class="jumbotron">
        <h1>DURGASOFT ONLINE SHOPPING APP</h1>
        <a class="btn btn-primary btn-lg" href="/add" role="button">ADD ITEM</a>
        <a class="btn btn-primary btn-lg" href="/display" role="button">Display ITEMS</a>
      </div>

    </div>
  </body>
</html>
```

<u>additem.html:</u>

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
```

```html
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
  </head>
  <body>
    <div class="container" align='center'>
      <h1>Add Item Form</h1>
      <form method="post">
        {{form.as_p}}
        {%csrf_token%}
        <input type="submit" name="" value="Add Item">
      </form><br><br><br>
      <a class="btn btn-primary btn-lg" href="/display" role="button">Display ITEMS</a>
    </div>
  </body>
</html>
```

<u>showitems.html:</u>

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Total Cookies Information:</h1>
    {%if request.COOKIES %}
    <table border=2>
      <thead>
        <th>Cookie Name</th>
        <th>Cookie Value</th>
      </thead>

      {% for key,value in request.COOKIES.items %}
      <tr>
        <td>{{key}}</td>
        <td>{{value}}</td>
      </tr>
      {% endfor %}
      </table>
     {%else%}
     <p>Cookie Information is not available</p>
    {%endif%}
</body>
</html>
```

Session Management By using Session API:

(Django Session Framework)

Diagram

Once client sends request to the server,if server wants to remember client information for the future purpose then server will create session object and store required information in that object. For every session object a unique identifier available which is nothing but sessionid. Server sends the corresponding session id to the client as the part of response. Client retrieves the session id from the response and save in the local file system. With every consecutive request client will that session id. By accessing that session id and corresponding session object server can remember client. This mechanism is nothing but session management by using session api.

Note: Session information will be stored in one of following possibilities

1. Inside a File
2. Inside a database
3. Inside Cache

The most straight forward approach is to use django.contrib.sessions application to store session information in a Django Model/database.

The Model Name is:
django.contrib.sessions.models.Session

Note: To use this approach compulsory the following applicaiton should be configured inside INSTALLED_APPS list of settings.py file.

```
    django.contrib.sessions
```

If it is not there then we can add,but we have to synchronize database

```
   python manage.py syncdb
```

Note:

```
INSTALLED_APPS = [
   ....
   'django.contrib.sessions',
```

```
    ...
]

MIDDLEWARE = [
    ..
    'django.contrib.sessions.middleware.SessionMiddleware',
    ....
]
```

Useful Methods for session Management:

1. request.session['key']=value
   To add data to the session

2. value=request.session['key']
   To get data from the session

3. request.session.set_expiry(seconds)
   Sets the expiry time for the session

4. request.session.get_expiry_age()
   returns the expiry age in seconds(the number of seconds until this session expire)

5. request.session.get_expiry_date()
   Returns the data on which this session will expire

Note: Before using session object in our application,compulsory we have to migrate. otherwise we will get the following error.
no such table: django_session

Session Demo1:

views.py:

```python
from django.shortcuts import render

# Create your views here.
def page_count_view(request):
    count=request.session.get('count',0)
    newcount=count+1
    request.session['count']=newcount
    print(request.session.get_expiry_age())
    print(request.session.get_expiry_date())
    return render(request,'testapp/pagecount.html',{'count':newcount})
```

pagecount.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <style >
    span{
      font-size: 300px;
    }

    </style>
  </head>
  <body>
    <h1>The Page Count:<span>{{count}}</span></h1>
  </body>
</html>
```