Django emphasizes reusability of components, also referred to as DRY (Don't Repeat Yourself), and comes with ready-to-use features like login system, database connection and CRUD operations (Create Read Update Delete).

Model: The model provides data from the database. The models are usually located in a file called models.py.
View: A view is a function or method that takes http requests as arguments, imports the relevant model(s), and finds out what data to send to the template, and returns the final result. The views are usually located in a file called views.py.
Template: A template is a file where you describe how the result should be represented. Templates are often .html files.
URLs: When a user requests a URL, Django decides which view it will send it to. This is done in a file called urls.py.

When you have installed Django and created you first Django web application, and the browser requests the URL, this is basically what happens:
1. Django receives the URL, checks the urls.py file, and calls the view that matches the URL.
2. The view, located in views.py, checks for relevant models.
3. The models are imported from the models.py file.
4. The view then sends the data to a specified template in the template folder.
5. The template contains HTML and Django tags, and with the data it returns finished HTML content back to the browser.

Virtual Environment
It is suggested to have a dedicated virtual environment for each Django project, and one way to manage a virtual environment is venv, which is included in Python.
py -m venv myproject

Then you have to activate the environment, by typing this command
myproject\Scripts\activate.bat
Note: You must activate the virtual environment every time you open the command prompt to work on your project.

Install Django
py -m pip install Django

Check Django Version
django-admin --version
Now you are ready to create a Django project in a virtual environment on your computer.

Django Create Project
Once you have come up with a suitable name for your Django project, like mine: myworld, navigate to where in the file system you want to store the code (in the virtual environment "myproject"), and run this command in the command prompt:

django-admin startproject myworld

Run the Django Project
Now that you have a Django project, you can run it, and see what it looks like in a browser.
Navigate to the /myworld folder and execute this command in the command prompt:
py manage.py runserver
Open a new browser window and type 127.0.0.1:8000 in the address bar.

The result:

What's Next?
We have a Django project! The next step is to make an app in your project.
You cannot have a web page created with Django without an app.
An app is a web application that has a specific meaning in your project, like a home page, a contact form, or a members database. In this tutorial we will create an app that allows us to list and register members in a database.
Create App
I will name my app members. Start by navigating to the selected location where you want to store the app, and run the command below.
py manage.py startapp members

```
myworld
   manage.py
   myworld/
   members/
      migrations/
         __init__.py
      __init__.py
      admin.py
      apps.py
      models.py
      tests.py
      views.py
```

Views
Django views are Python functions that takes http requests and returns http response, like HTML documents. A web page that uses Django is full of views with different tasks and missions.
There is a views.py in your members folder that looks like this:
members/views.py:

```
from django.shortcuts import render
# Create your views here.
```

Find it and open it, and replace the content with this:

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

def index(request):
    return HttpResponse("Hello world!")
```

Well, we must call the view via a URL:
Create a file named urls.py in the same folder as the views.py file, and type this code in it:
members/urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

The urls.py file you just created is specific for the members application. We have to do some routing in the root directory myworld as well.
There is a file called urls.py on the myworld folder, open that file and add the include module in the import statement, and also add a path() function in the urlpatterns[] list, with arguments that will route users that comes in via 127.0.0.1:8000/members/.

Then your file will look like this:
myworld/urls.py:

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('members/', include('members.urls')),
    path('admin/', admin.site.urls),
]
```

If the server is not running, navigate to the /myworld folder and execute this command in the command prompt:

```
py manage.py runserver
```

Django Templates
Create a templates folder inside the members folder, and create a HTML file named myfirst.html.

```
myworld
    manage.py
    myworld/
```

```
    members/
        templates/
            myfirst.html
```

Open the HTML file and insert the following:
members/templates/myfirst.html:

```
<!DOCTYPE html>
<html>
<body>

<h1>Hello World!</h1>
<p>Welcome to my first Django project!</p>

</body>
</html>
```

Modify the View
Open the views.py file and replace the index view with this:
members/views.py:

```
from django.http import HttpResponse
from django.template import loader

def index(request):
  template = loader.get_template('myfirst.html')
  return HttpResponse(template.render())
```

Change Settings
Look up the INSTALLED_APPS[] list and add the members app like this:
myworld/settings.py:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'members.apps.MembersConfig'
]
```

Then run this command:

```
py manage.py migrate
py manage.py runserver
```

In the browser window, type 127.0.0.1:8000/members/ in the address bar.

The result should look like this:

Django Models
A Django model is a table in your database.
When we created the Django project, we got an empty SQLite database. It was created in the myworld root folder.

Create Table (Model)
To create a new table, we must create a new model.
To add a Members table in our database, start by creating a Members class, and describe the table fields in it:
members/models.py:

```
from django.db import models
class Members(models.Model):
  firstname = models.CharField(max_length=255)
  lastname = models.CharField(max_length=255)
```

Then navigate to the /myworld/ folder and run this command:
py manage.py makemigrations members

Django creates a file with any new changes and stores the file in the /migrations/ folder.
Next time you run py manage.py migrate Django will create and execute an SQL statement, based on the content of the new file in the migrations folder.
py manage.py migrate

The SQL statement created from the model is:

```
CREATE TABLE "members_members" (
"id" INT NOT NULL PRIMARY KEY AUTOINCREMENT,
"firstname" varchar(255) NOT NULL,
"lastname" varchar(255) NOT NULL);
```

Django Add Members
You will learn how to make a user interface that will take care of CRUD operations (Create, Read, Update, Delete), but for now, let's write Python code directly in the Python interpreter (Python shell) and add some members in our database, without the user interface. To open a Python shell, type this command:
py manage.py shell
At the bottom, after the three >>> write the following:

```
>>> from members.models import Members
```

Write this to look at the empty Members table:

```
>>> Members.objects.all()
```

This should give you an empty QuerySet object, like this:

```
<QuerySet []>
```

A QuerySet is a collection of data from a database.

Add a record to the table, by executing these two lines:

```
>>> member = Members(firstname='Emil', lastname='Refsnes')
>>> member.save()
```

Execute this command to see if the Members table got a member:

```
>>> Members.objects.all().values()
```

Hopefully, the result will look like this:

```
<QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'}]>
```

Add Multiple Records

```
>>> member1 = Members(firstname='Tobias', lastname='Refsnes')
>>> member2 = Members(firstname='Linus', lastname='Refsnes')
>>> member3 = Members(firstname='Lene', lastname='Refsnes')
>>> member4 = Members(firstname='Stale', lastname='Refsnes')
>>> members_list = [member1, member2, member3, member4]
>>> for x in members_list:
>>>   x.save()
```

```
>>> Members.objects.all().values()
<QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'},
{'id': 2, 'firstname': 'Tobias', 'lastname': 'Refsnes'}, {'id': 3, 'firstname': 'Linus', 'lastname':
'Refsnes'}, {'id': 4, 'firstname': 'Lene', 'lastname': 'Refsnes'}, {'id': 5, 'firstname': 'Stale', 'lastname':
'Refsnes'}]>
```

To see the result in a web page, we can create a view for this particular task.
In the members app, open the views.py file, it should look like this:

```
from django.http import HttpResponse
from django.template import loader

def index(request):
  template = loader.get_template('myfirst.html')
  HttpResponse(template.render())
```

Change the content in the views.py file to look like this instead:

```
from django.http import HttpResponse
from django.template import loader
from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
  output = ""
  for x in mymembers:
    output += x["firstname"]
  return HttpResponse(output)
```

See the result in your browser. If you are still in the Python shell, write this command to exit the shell:

>>> quit()
Navigate to the /myworld/ folder and type this to start the server:

py manage.py runserver
In the browser window, type 127.0.0.1:8000/members/ in the address bar.

The result:

Adding Template
We have managed to display the content of a database table in a web page. To add some HTML around the values, we will create a template for the application.

All templates must be located in the templates folder off your app, if you have not already created a templates folder, do it now.

In the templates folder, create a file named index.html, with the following content:
members/templates/index.html:

```
<h1>Members</h1>
<table border="1">
{% for x in mymembers %}
<tr>
<td>{{ x.id }}</td>
<td>{{ x.firstname }}</td>
<td>{{ x.lastname }}</td>
</tr>
{% endfor %}
</table>
```

Did you notice the {% %} and {{ }} parts? They are called template tags. Template tags allows you to perform logic and render variables in your templates.
Modify the View
Change the index view to include the template:
members/views.py:

```
from django.http import HttpResponse
from django.template import loader
from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
```

```
  template = loader.get_template('index.html')
  context = {
    'mymembers': mymembers,
  }
  return HttpResponse(template.render(context, request))
```

In the browser window, type 127.0.0.1:8000/members/ in the address bar.
The result:

Adding Records
So far we have created a Members table in our database, and we have inserted five records by writing code in the Python shell. Now we want to be able to create new members from a web page.
Start by adding a link in the members template:
members/templates/index.html:

```
<h1>Members</h1>

<table border="1">
{% for x in mymembers %}
<tr>
<td>{{ x.id }}</td>
<td>{{ x.firstname }}</td>
<td>{{ x.lastname }}</td>
</tr>
{% endfor %}
</table>

<p>
<a href="add/">Add member</a>
</p>
```

Add a new template in the templates folder, named add.html:
members/templates/add.html:

```
<h1>Add member</h1>

<form action="addrecord/" method="post">
{% csrf_token %}
First Name:<br>
<input name="first">
<br><br>
Last Name:<br>
<input name="last">
<br><br>
<input type="submit" value="Submit">
```

```
</form>
```
Note: Django requires this line in the form: {% csrf_token %} to handle Cross Site Request Forgeries in forms where the method is POST.
Next, add a view in the members/views.py file, name the new view add:
members/views.py:
```
from django.http import HttpResponse
from django.template import loader
from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
  template = loader.get_template('index.html')
  context = {
    'mymembers': mymembers,
  }
  return HttpResponse(template.render(context, request))

def add(request):
  template = loader.get_template('add.html')
  return HttpResponse(template.render({}, request))
```
URLs
Add a path() function in the members/urls.py file, that points the url 127.0.0.1:8000/members/add/ to the right location:

members/urls.py:
```
from django.urls import path
from . import views

urlpatterns = [
  path('', views.index, name='index'),
  path('add/', views.add, name='add'),
]
```
In the browser, click the "Add member" link and the result should look like this:

The action attribute specifies where to send the form data, in this case the form data will be sent to addrecord/, so we must add a path() function in the members/urls.py file that points to the right view:

members/urls.py:
```
from django.urls import path
from . import views

urlpatterns = [
```

```
  path('', views.index, name='index'),
  path('add/', views.add, name='add'),
  path('add/addrecord/', views.addrecord, name='addrecord'),
]
```

Code for Adding Records

Make sure you add the addrecord view in the in the members/views.py file:

```
from django.http import HttpResponse, HttpResponseRedirect
from django.template import loader
from django.urls import reverse
from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
  template = loader.get_template('index.html')
  context = {
    'mymembers': mymembers,
  }
  return HttpResponse(template.render(context, request))

def add(request):
  template = loader.get_template('add.html')
  return HttpResponse(template.render({}, request))

def addrecord(request):
  x = request.POST['first']
  y = request.POST['last']
  member = Members(firstname=x, lastname=y)
  member.save()
  return HttpResponseRedirect(reverse('index'))
```

The addrecord view does the following:

- Gets the first name and last name with the request.POST statement.
- Adds a new record in the members table.
- Redirects the user back to the index view.

Try to add your names into the members' table:

Deleting Records

Add a "delete" column in the members template:

members/templates/index.html:

```
<h1>Members</h1>
```

```
<table border="1">
{% for x in mymembers %}
  <tr>
  <td>{{ x.id }}</td>
  <td>{{ x.firstname }}</td>
  <td>{{ x.lastname }}</td>
  <td><a href="delete/{{ x.id }}">delete</a></td>
  </tr>
{% endfor %}
</table>

<p>
<a href="add/">Add member</a>
</p>
```

The "delete" link in the HTML table points to 127.0.0.1:8000/members/delete/ so we will add a path() function in the members/urls.py file, that points the url to the right location, with the ID as a parameter:

members/urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
  path('', views.index, name='index'),
  path('add/', views.add, name='add'),
  path('add/addrecord/', views.addrecord, name='addrecord'),
  path('delete/<int:id>', views.delete, name='delete'),
]
```

Now we need to add a new view called delete in the members/views.py file:

```
from django.http import HttpResponse, HttpResponseRedirect
from django.template import loader
from django.urls import reverse

from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
  template = loader.get_template('index.html')
  context = {
    'mymembers': mymembers,
```

```
  }
  return HttpResponse(template.render(context, request))

def add(request):
  template = loader.get_template('add.html')
  return HttpResponse(template.render({}, request))

def addrecord(request):
  x = request.POST['first']
  y = request.POST['last']
  member = Members(firstname=x, lastname=y)
  member.save()
  return HttpResponseRedirect(reverse('index'))

def delete(request, id):
  member = Members.objects.get(id=id)
  member.delete()
  return HttpResponseRedirect(reverse('index'))
```

The delete view does the following:
- Gets the id as an argument.
- Uses the id to locate the correct record in the Members table.
- Deletes that record.
- Redirects the user back to the index view.

Updating Records:
Start by adding a link for each member in the table:
members/templates/index.html:

```
<h1>Members</h1>

<table border="1">
{% for x in mymembers %}
<tr>
<td><a href="update/{{ x.id }}">{{ x.id }}</a></td>
<td>{{ x.firstname }}</td>
<td>{{ x.lastname }}</td>
<td><a href="delete/{{ x.id }}">delete</a>
</tr>
{% endfor %}
</table>

<p>
<a href="add/">Add member</a>
</p>
```

The link goes to a view called update with the ID of the current member.
The result will look like this:

Next, add the update view in the members/views.py file:
members/views.py:

```
from django.http import HttpResponse, HttpResponseRedirect
from django.template import loader
from django.urls import reverse
from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
  template = loader.get_template('index.html')
  context = {
    'mymembers': mymembers
  }
  return HttpResponse(template.render(context, request))

def add(request):
  template = loader.get_template('add.html')
  return HttpResponse(template.render({}, request))

 def addrecord(request):
  first = request.POST['first']
  last = request.POST['last']
  member = Members(firstname=first, lastname=last)
  member.save()

return HttpResponseRedirect(reverse('index'))

def delete(request, id):
  member = Members.objects.get(id=id)
  member.delete()
  return HttpResponseRedirect(reverse('index'))

def update(request, id):
  mymember = Members.objects.get(id=id)
  template = loader.get_template('update.html')
  context = {
    'mymember': mymember,
  }
  return HttpResponse(template.render(context, request))
```

The update view does the following:
- Gets the id as an argument.
- Uses the id to locate the correct record in the Members table.
- loads a template called update.html.
- Creates an object containing the member.
- Sends the object to the template.
- Outputs the HTML that is rendered by the template.

New Template
Add a new template in the templates folder, named update.html:
members/templates/update.html:

```
<h1>Update member</h1>

<form action="updaterecord/{{ mymember.id }}" method="post">
{% csrf_token %}
First Name:<br>
<input name="first" value="{{ mymember.firstname }}">
<br><br>
Last Name:<br>
<input name="last" value="{{ mymember.lastname }}">
<br><br>
<input type="submit" value="Submit">
</form>
```

Add a path() function in the members/urls.py file, that points the url 127.0.0.1:8000/members/update/ to the right location, with the ID as a parameter:
members/urls.py:

```
from django.urls import path

from . import views

urlpatterns = [
  path('', views.index, name='index'),
  path('add/', views.add, name='add'),
  path('add/addrecord/', views.addrecord, name='addrecord'),
  path('delete/<int:id>', views.delete, name='delete'),
  path('update/<int:id>', views.update, name='update'),
]
```

What Happens on Submit?

Did you notice the action attribute in the HTML form? The action attribute specifies where to send the form data, in this case the form data will be sent to: updaterecord/{{ mymember.id }}, so we must add a path() function in the members/urls.py file that points to the right view:
members/urls.py:

```
from django.urls import path

from . import views

urlpatterns = [
  path('', views.index, name='index'),
  path('add/', views.add, name='add'),
  path('add/addrecord/', views.addrecord, name='addrecord'),
  path('delete/<int:id>', views.delete, name='delete'),
  path('update/<int:id>', views.update, name='update'),
  path('update/updaterecord/<int:id>', views.updaterecord, name='updaterecord'),
]
```

Make sure you add the updaterecord view in the in the members/views.py file:
members/views.py:

```
from django.http import HttpResponse, HttpResponseRedirect
from django.template import loader
from django.urls import reverse
from .models import Members

def index(request):
  mymembers = Members.objects.all().values()
  template = loader.get_template('index.html')
  context = {
    'mymembers': mymembers,
  }
  return HttpResponse(template.render(context, request))

def add(request):
  template = loader.get_template('add.html')
  return HttpResponse(template.render({}, request))

def addrecord(request):
  x = request.POST['first']
  y = request.POST['last']
  member = Members(firstname=x, lastname=y)
  member.save()
  return HttpResponseRedirect(reverse('index'))
```

```
def delete(request, id):
  member = Members.objects.get(id=id)
  member.delete()
  return HttpResponseRedirect(reverse('index'))

def update(request, id):
  mymember = Members.objects.get(id=id)
  template = loader.get_template('update.html')
  context = {
    'mymember': mymember,
  }
  return HttpResponse(template.render(context, request))

def updaterecord(request, id):
  first = request.POST['first']
  last = request.POST['last']
  member = Members.objects.get(id=id)
  member.firstname = first
  member.lastname = last
  member.save()
  return HttpResponseRedirect(reverse('index'))
```

The updaterecord function will update the record in the members table with the selected ID. Try to update a record and see how it works:

Template Variables

In Django templates, you can render variables by putting them inside {{ }} brackets:

Example - template.html:

```
<h1>Hello {{ firstname }}, how are you?</h1>
```

Create Variable in View

The variable firstname in the example above was sent to the template via a view:

views.py:

```
from django.http import HttpResponse
from django.template import loader

def testing(request):
  template = loader.get_template('template.html')
  context = {
    'firstname': 'Linus',
  }
```

```
  return HttpResponse(template.render(context, request))
```
As you can see in the view above, we create an object named context and fill it with data, and send it as the first parameter in the template.render() function.

Create Variables in Template
You can also create variables directly in the template, by using the {% with %} template tag:
Example - template.html:

```
{% with firstname="Tobias" %}
<h1>Hello {{ firstname }}, how are you?</h1>
```

Example of if else statements:
```
{% if greeting == 1 %}
  <h1>Hello</h1>
{% elif greeting == 2 %}
  <h1>Welcome</h1>
{% else %}
  <h1>Goodbye</h1>
{% endif %}
```
Note: no colon (:) to close the if or else statement. We use endif at the end.

For Loops
A for loop is used for iterating over a sequence, like looping over items in an array, a list, or a dictionary.
Loop through items fetched from a database:
```
{% for x in members %}
  <h1>{{ x.id }}</h1>
  <p>
    {{ x.firstname }}
    {{ x.lastname }}
  </p>
{% endfor %}
```

Reversed
The reversed keyword is used when you want to do the loop in reversed order.
```
{% for x in members reversed %}
  <h1>{{ x.id }}</h1>
  <p>
    {{ x.firstname }}
    {{ x.lastname }}
  </p>
{% endfor %}
```

Empty

The empty keyword can be used if you want to do something special if the object is empty.

```
<ul>
  {% for x in emptytestobject %}
    <li>{{ x.firstname }}</li>
  {% empty %}
    <li>No members</li>
  {% endfor %}
</ul>
```

The empty keyword can also be used if the object does not exist:

```
<ul>
  {% for x in myobject %}
    <li>{{ x.firstname }}</li>
  {% empty %}
    <li>No members</li>
  {% endfor %}
</ul>
```

Loop Variables

Django has some variables that are available for you inside a loop:

- forloop.counter
- forloop.counter0
- forloop.first
- forloop.last
- forloop.parentloop
- forloop.revcounter
- forloop.revcounter0
- forloop.counter

The current iteration, starting at 1.

```
<ul>
  {% for x in fruits %}
    <li>{{ forloop.counter }}</li>
  {% endfor %}
</ul>
```

forloop.first

Allows you to test if the loop is on its first iteration.

Example

Draw a blue background for the first iteration of the loop:

```
<ul>
  {% for x in fruits %}
    <li
      {% if forloop.first %}
```

```
      style='background-color:lightblue;'
    {% endif %}
  >{{ x }}</li>
 {% endfor %}
</ul>
```

Draw a blue background for the last iteration of the loop:

```
<ul>
 {% for x in fruits %}
  <li
    {% if forloop.last %}
      style='background-color:lightblue;'
    {% endif %}
  >{{ x }}</li>
 {% endfor %}
</ul>
```

forloop.revcounter
The current iteration if you start at the end and count backwards, ending up at 1.

```
<ul>
 {% for x in fruits %}
  <li>{{ forloop.revcounter }}</li>
 {% endfor %}
</ul>
```

forloop.revcounter0
The current iteration if you start at the end and count backwards, ending up at 0.

```
<ul>
 {% for x in fruits %}
  <li>{{ forloop.revcounter0 }}</li>
 {% endfor %}
</ul>
```

Comments
Comments allows you to have sections of code that should be ignored.

```
<h1>Welcome Everyone</h1>
{% comment %}
 <h1>Welcome ladies and gentlemen</h1>
{% endcomment %}
```

You can add a message to your comment, to help you remember why you wrote the comment, or as message to other people reading the code.

```
<h1>Welcome Everyone</h1>
{% comment "this was the original welcome message" %}
   <h1>Welcome ladies and gentlemen</h1>
{% endcomment %}
```

Smaller Comments
You can also use the {# ... #} tags when commenting out code, which can be easier when for smaller comments:

```
<h1>Welcome{# Everyone#}</h1>
```

Comment in Views
Views are written in Python, and Python comments are written with the # character.

Django cycle Tag
The cycle tag allows you to do different tasks for different iterations. The cycle tag takes arguments, the first iteration uses the first argument, the second iteration uses the second argument etc.
If you want to have a new background color for each iteration, you can do that with the cycle tag:

```
<ul>
 {% for x in members %}
   <li style='background-color:{% cycle 'lightblue' 'pink' 'yellow' 'coral' 'grey' %}'>
     {{ x.firstname }}
   </li>
 {% endfor %}
</ul>
```

If the cycle reaches the end of the arguments, it starts over:

```
<ul>
 {% for x in members %}
   <li style='background-color:{% cycle 'lightblue' 'pink' %}'>
     {{ x.firstname }}
   </li>
 {% endfor %}
</ul>
```

Restart the cycle after 3 cycles:

```
<ul>
 {% for x in members %}
   {% cycle 'lightblue' 'pink' 'yellow' 'coral' 'grey' as bgcolor silent %}
   {% if forloop.counter == 3 %}
     {% resetcycle %}
   {% endif %}
   <li style='background-color:{{ bgcolor }}'>
```

```
    {{ x.firstname }}
   </li>
 {% endfor %}
</ul>
```

Extends
The extends tag allows you to add a parent template for the current template. This means that you can have one master page that acts like a parent for all other pages:
Example
mymaster.html:

```
<html>
<body>

<h1>Welcome</h1>

{% block mymessage %}
{% endblock %}

</body>
</html>
```

template.html:

```
{% extends 'mymaster.html' %}

{% block mymessage %}
  <p>This page has a master page</p>
{% endblock %}
```

You put placeholders in the master template, telling Django where to put which content.

Include
The include tag allows you include a template inside the current template. This is useful when you have a block of content that are the same for many pages.

Example
footer.html:

```
<p>You have reach the bottom of this page, thank you for your time.</p>
```

template.html:

```
<h1>Hello</h1>

<p>This page contains a footer in a template.</p>

{% include 'footer.html' %}
```

Filter a Value

With the pipe | character followed by a filter name, you can run a value through a filter before returning it. The name of the filter defines what the filter will do with the value.
Example:
Return the variable firstname with upper case letters:

```
<h1>Hello {{ firstname|upper }}, how are you?</h1>
```

The filter Tag
The filter tag allows you to run a whole section of code through a filter, and return it according to the filter keyword(s).
Example:
Return the variable firstname with upper case letters:

```
{% filter upper %}
  <h1>Hello everyone, how are you?</h1>
{% endfilter %}
```

Add a Static CSS File
When building web applications, you probably want to add some static files like images or css files. Start by creating a folder named static in your project, the same place where you created the templates folder:
Add a file in the static folder, name it myfirst.css:
members/static/myfirst.css:

```
body {
  background-color: lightblue;
  font-family: verdana;
}
```

Now you have a css file, with some css properties that will style an HTML page. The next step will be to include this file in a HTML template:

Open the HTML file and add the following:

```
{% load static %}
```

And:

```
<link rel="stylesheet" href="{% static 'myfirst.css' %}">
```

Restart the server for the changes to take effect:

```
py manage.py runserver
```

members/templates/template.html:

```
{% load static %}
<!DOCTYPE html>
<html>
<link rel="stylesheet" href="{% static 'myfirst.css' %}">
<body>
```

```
{% for x in fruits %}
  <h1>{{ x }}</h1>
{% endfor %}

</body>
</html>
```

Note: For some reason, make sure that DEBUG = True in the settings.py file.

Add a Static JS File

Adding js files in Django project is done exactly the same way as adding css files in Django: Static files, like css, js, and images, goes in the static folder. If you do not have one, create it in the same location as you created the templates folder:

Add a js file in the static folder, name it myfirst.js:

members/static/myfirst.js:

```
function myFunction() {
  alert("Hello from a static file!");
}
```

Restart the server for the changes to take effect:

```
py manage.py runserver
```

```
{% load static %}
<!DOCTYPE html>
<html>
<script src="{% static 'myfirst.js' %}"></script>
<body>

<button onclick="myFunction()">Click me!</button>

</body>
</html>
```

Order By

To sort QuerySets, Django uses the order_by() method:

Example

Order the the result alphabetically by firstname:

```
mydata = Members.objects.all().order_by('firstname').values()
```

Descending Order

By default, the result is sorted ascending (the lowest value first), to change the direction to descending (the highest value first), use the minus sign (NOT), - in front of the field name:

Example

Order the the result firstname descending:

```
mydata = Members.objects.all().order_by('-firstname').values()
```