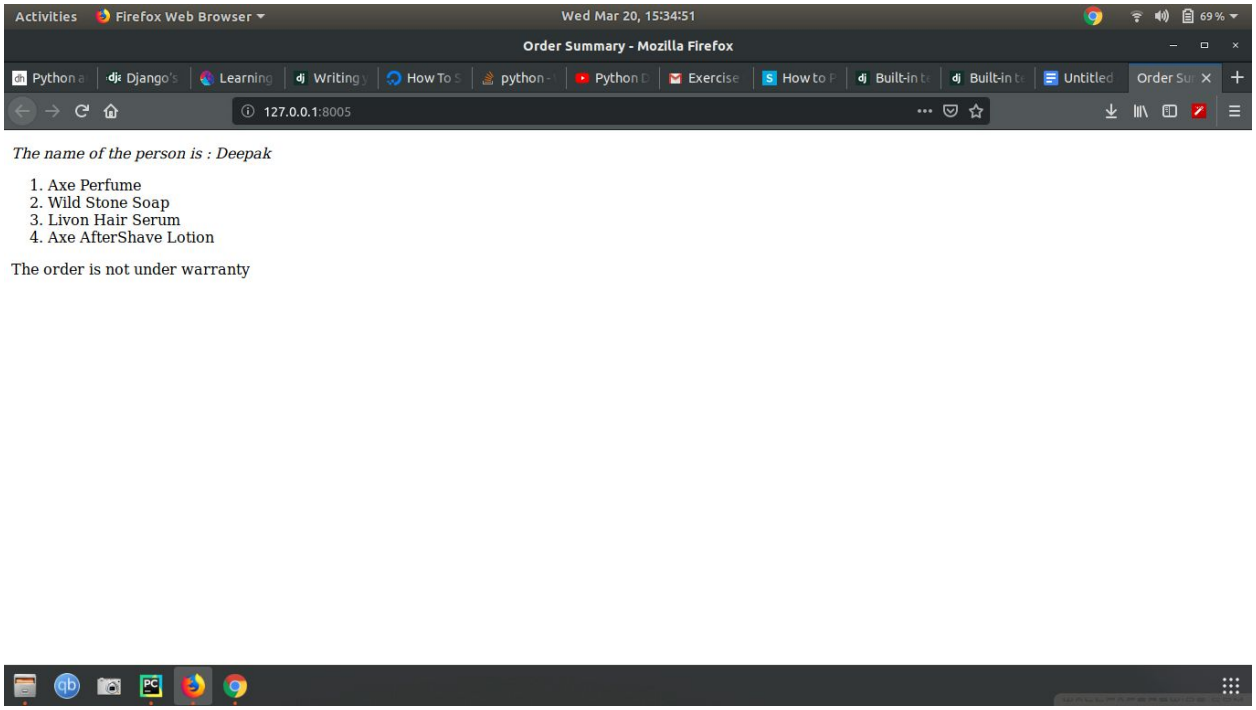
1. render a html file order.html and pass a variable "person_name", an array of order_list containing the item names, and a conditional flag "ordered_warranty" and display it.

Ans



https://github.com/tripathideepak1997/DjangoUmderstand

2. what is render() ?
Ans

render(*request*, *template_name*, *context=None*, *content_type=None*, *status=None*, *using=None*)

Combines a given template with a given context dictionary and returns an HttpResponse object with that rendered text.

Django does not provide a shortcut function which returns a TemplateResponse because the constructor of TemplateResponse offers the same level of convenience as render().

3. difference between the tag {% include %}. and {% extends %}
Ans

Extending allows you to replace blocks (e.g. "content") from a parent template instead of including parts to build the page (e.g. "header" and "footer"). This allows you to have a single

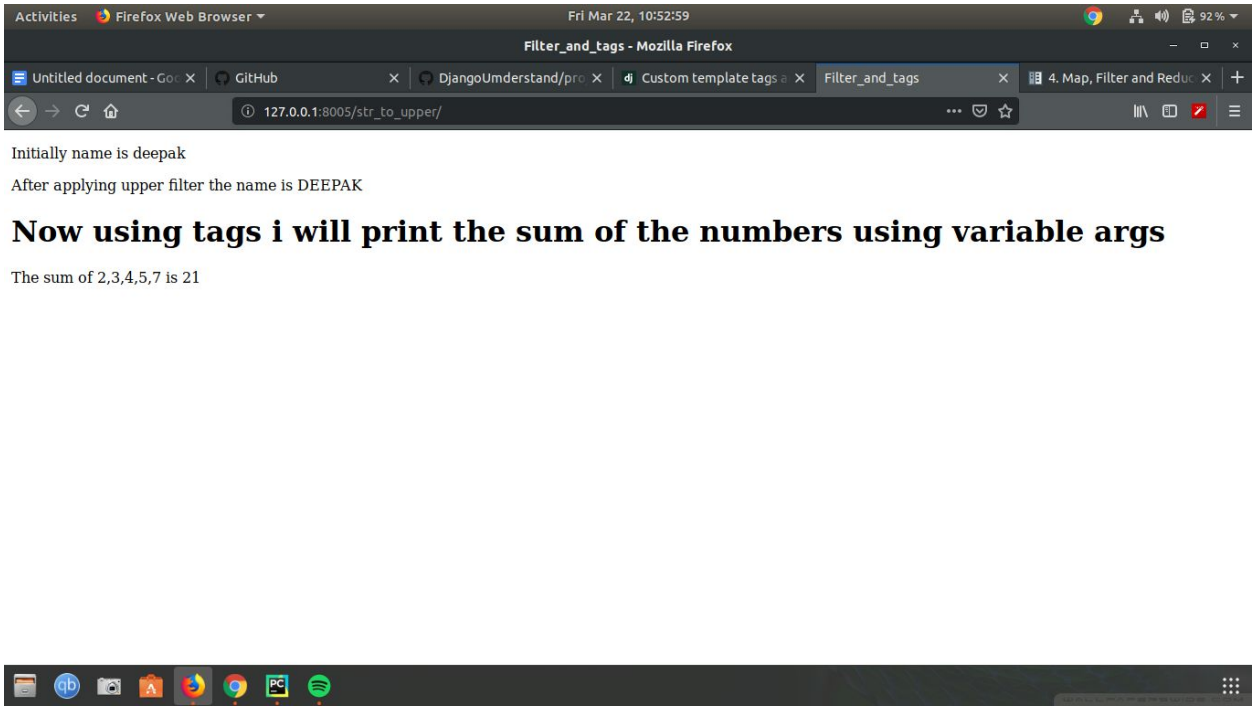
template containing your complete layout and you only "insert" the content of the other template by replacing a block.
If the user profile is used on all pages, you'd probably want to put it in your base template which is extended by others or include it into the base template. If you wanted the user profile only on very few pages, you could also include it in those templates. If the user profile is the same except on a few pages, put it in your base template inside a block which can then be replaced in those templates which want a different profile.

4. write steps to create the custom tags and filter, give one working example of each.
Ans

- Add templatetags directory in your app
- Now add two files in it : __init__.py and one file that has some tags and filters functions in it like tags_and_filters.py
- Now in that file add your custom filter and tag functionality using functions in python like **def to_upper(string_to_be_converted)** as my filter function and **def add(*args)** as my tag function
- Add this line in that file **from django import template**
  register = template.Library()
- Now register the tags and filters using decorators in that file

```
@register.filter(name='UPPER')
def to_upper(inp_string):
   return inp_string.upper()

@register.simple_tag(name='doing_arithmetic')
def add(*args):
   return reduce(lambda x, y: x + y, args)
```

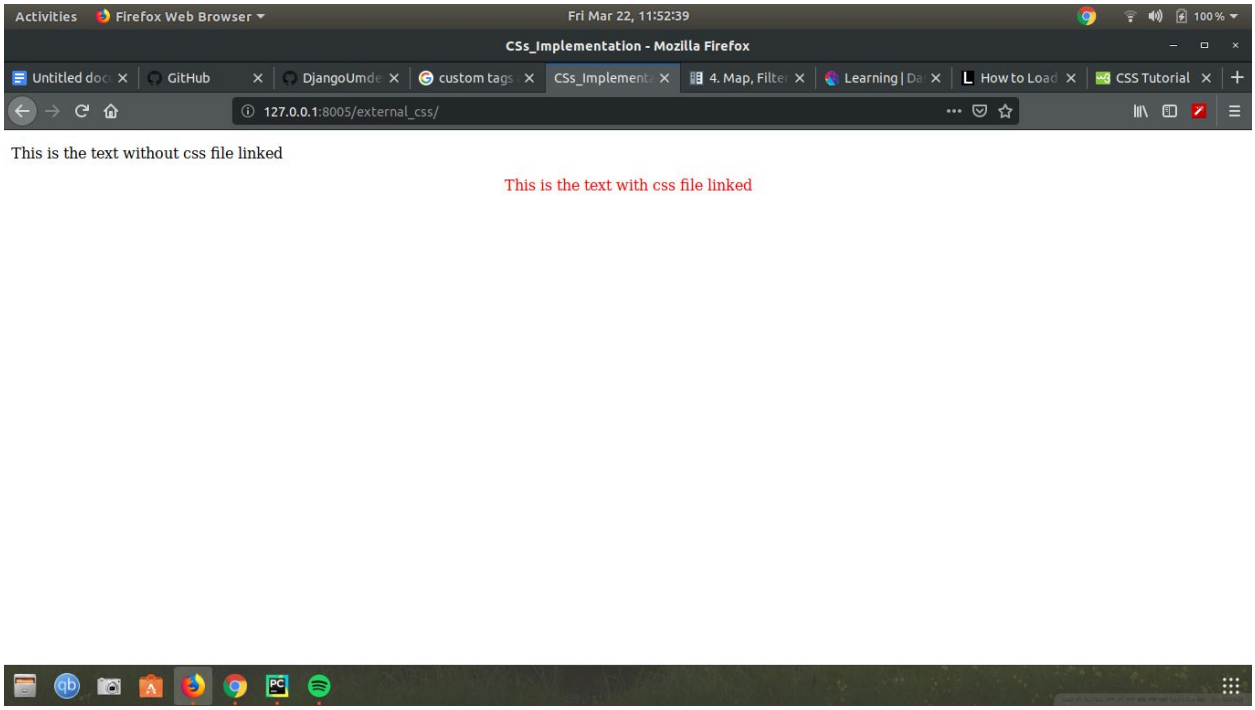
- Add {% load tags_and_filters %}
  In your html file



5. write steps for setting up the statics files like .css or .js file, give one working example.
Ans

- Thus, to load CSS files in a template in Django, we add the code to the head tag. The code that we add to the head tag is shown below.

```
{% load static %}
<head>

  <link rel="stylesheet" href="{% static 'file1.css' %}">
  <meta charset="UTF-8">
  <title>CSs_Implementation</title>
</head>
```



Static_files.html

```
<!DOCTYPE html>
<html lang="en">
{% load static %}
<head>

  <link rel="stylesheet" href="{% static 'file1.css' %}">
  <meta charset="UTF-8">
  <title>CSs_Implementation</title>
</head>
<body>
  <p>This is the text without css file linked</p>
  <p class="a" > This is the text with css file linked </p>


</body>
</html>
```

File1.html

```
.a
{
 color: red;
 text-align: center;
}
```

Views.py

```
from django.shortcuts import render
def external_css(request):
   return render(request,'static_files.html')
```

6. What is Pagination ? create a list of 1000 integers then using this list apply pagination.
Ans
Django provides a few classes that help you manage paginated data – that is, data that's split across several pages, with "Previous/Next" links. These classes live in django/core/paginator.py.

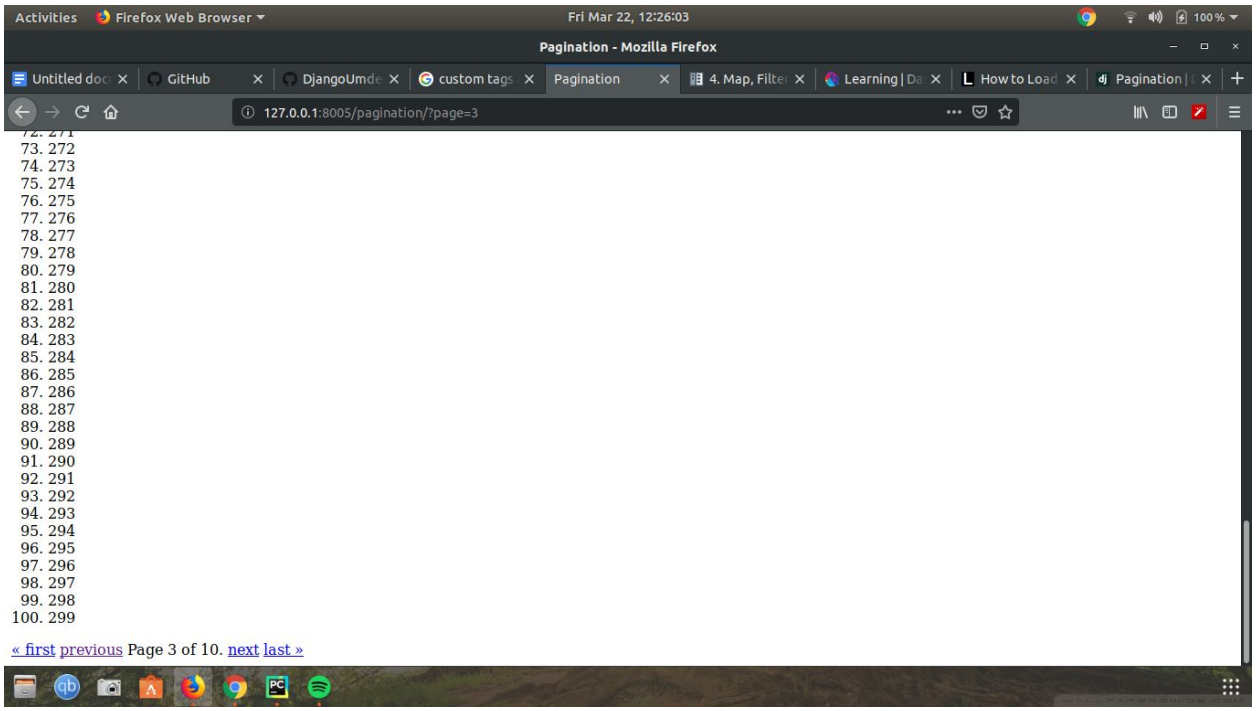
Pagination.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Pagination</title>
</head>
<body>
   <ol>
      {% for contact in contacts %}
      <li>{{contact}}</li>
      {% endfor %}

   </ol>

<div class="pagination">
   <span class="step-links">
      {% if contacts.has_previous %}
         <a href="?page=1">&laquo; first</a>
         <a href="?page={{ contacts.previous_page_number }}">previous</a>
      {% endif %}

      <span class="current">
         Page {{ contacts.number }} of {{ contacts.paginator.num_pages }}.
      </span>

      {% if contacts.has_next %}
         <a href="?page={{ contacts.next_page_number }}">next</a>
         <a href="?page={{ contacts.paginator.num_pages }}">last &raquo;</a>
      {% endif %}
   </span>
</div>
```

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

Views.py

```
from django.core.paginator import Paginator
from django.shortcuts import render


def do_pagination(request):
    list1 = [i for i in range(1000)]
    paginator = Paginator(list1,100)
    page_no = request.GET.get('page')
    contacts = paginator.get_page(page_no)
    return render(request, 'pagination.html', {'contacts': contacts})
```



7. What is the variable BACKEND, DIRS, APP_DIRS, OPTIONS used for ?

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
```

```
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

Ans

A list containing the settings for all template engines to be used with Django. Each item of the list is a dictionary containing the options for an individual engine.

BACKEND
The template backend to use. The built-in template backends are:

- 'django.template.backends.django.DjangoTemplates'
- 'django.template.backends.jinja2.Jinja2'

You can use a template backend that doesn’t ship with Django by setting BACKEND to a fully-qualified path (i.e. 'mypackage.whatever.Backend').

DIRS
Directories where the engine should look for template source files, in search order.

APP_DIRS
Default: False
Whether the engine should look for template source files inside installed applications.

OPTIONS
Default: {} (Empty dict)
Extra parameters to pass to the template backend. Available parameters vary depending on the template backend.

DjangoTemplates engines accept the following OPTIONS:

'autoescape': a boolean that controls whether HTML autoescaping is enabled.
It defaults to True.

'context_processors': a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a dict of items to be merged into the context.
It defaults to an empty list.