

Python Learning Journal

Guy Rimel

About Myself

I'm just a normal Guy. I'm 32 years old, I live in Texas, I love Jesus, and my wife, and good coffee. I took a coding class in college (back in 2017) and it has piqued my interest ever since. Though, only in the last year have I gotten serious about software.

I enjoy the basic full-stack, and I'm a web-content creator for Texas Tech University, but making websites is simple stuff. Diving into packages, frameworks, backends, APIs, and more complex languages is equal parts fun and headache inducing.

1.1: Getting Started With Python

Learning Goals

- Summarize the uses and benefits of Python for web development
- Prepare your developer environment for programming with Python

Okay, starting off my Python journal here. Day one saw me installing Python with the Windows x64 installer. I'm running out of disk space on my laptop. The installer does it's thing, I open a command prompt and initiate python's built in REPL (Read Execute Print Loop) (is that pronounced "repple"?), then type in `print("Hello World!")` and just like that, I've mastered Python!

I learned about "environments" which are directories of particular Python packages and files to separate dependencies.

Another critical thing I learned was the *import* keyword, to utilize Python's built-in package management system, e.g., `import math` will bring in a package with mathematical functionality.

Similar to *import* is the keyword *pip* which installs packages from the Python Package Inventory. So, `pip install xyz` will install the xyz package in the current directory location from the terminal.

Python is dynamically typed, meaning that a variable can be assigned a string value, then reassigned as an integer value, with no qualms (like JavaScript).

Python lines do not have semicolons...

1.2: Data Types in Python

Learning Goals

- Explain variables and data types in Python
- Summarize the use of objects in Python
- Create a data structure for your Recipe app

Python data types are more fun than riding a jet ski. We all know that Python has data types of string, integer, boolean, tuple, list, dictionary, and noneType (and there are probably others).

It was good to practice the syntax of built-in functions and methods on different data types. How to “slice” properly, sort(), min(), max(), count(), pop(), extend(), append(), copy(), and more.

Apparently, objects in Python are called “dictionaries” allowing for an entity of key/value pairs of different data types.

For the basic data structures of the Achievement’s recipe app, I’ve chosen to store all recipes as a list of dictionaries (each recipe is a “dictionary”). This allows for standardized key/value pairs for all recipes.

1.3: Operators & Functions in Python

Learning Goals

- Implement conditional statements in Python to determine program flow
- Use loops to reduce time and effort in Python programming
- Write functions to organize Python code

Conditional statements execute code based on a boolean value. That’s it, that’s what they do: they execute a code block based on “True” or “False”. Start a conditional statement with the “if” keyword, then a value, then desired comparators, if any, like “AND” or “OR” or “NOT” and another value. End it with a colon. Indent. Now, write a code block to be executed if the statement is truthy. Now, add “elif” or “else” or perhaps *nest* another “if”.

The two loops discussed in this chapter were for-loops and while-loops. The for-loop will execute a code block for a certain number of times as specified by an “iterable”, which would be

anything with a “length” (array, tuple, string). The while-loop will run the code block FOREVER... until the conditional statement becomes falsy. If you said ```while True: ``` the code block would be executed in an infinite loop, and everyone would laugh and point at how terrible it is, forever.

In a loop, use “break” or “continue” to either exit the loop, or skip an iteration.

Functions are reusable code blocks that can be passed arguments to do or return something. Start a Python function with “def” to define the function, then the function name, then two parentheses “()” and if desired, throw some parameters in those parentheses. Parameters can be assigned default values with the equals sign.

1.4: File Handling in Python

Learning Goals

- Use files to store and retrieve data in Python

Using files to store and retrieve data in Python. This was a fresh new concept for me, but obviously a very important topic to create any meaningful Python-based software.

For basic (text) reading and writing, declare a file path and a method flag as arguments of the open function like so ``` file_variable = open('./thing.txt', 'rt') ``` the t is for “text” and it’s implied by default. Then use the `readlines()` method to read the data: ``` print(file_variable.readlines()) ``` to return an array of each line in the text file, and finally, CLOSE the file with the `.close()` method.

To READ a text file, you would use something like: ``` file_variable = open('./thing.txt', 'w') ``` then the method `.writelines(array_name)` would write each element of an iterable. But all the items would smoosh together, so be sure to include a newline character “\n” after each iteration.

Okay, now pickles. Use pickles to store more complex data types (dictionaries). First ``` import pickle ```, then, let’s say the dictionary is named “my_dictionary”, DUMP the dictionary into a .bin file like this: ``` file_variable = open('thing.bin', 'wb') ``` then ``` pickle.dump(my_dictionary, file_variable) ``` then ``` file_variable.close() ```

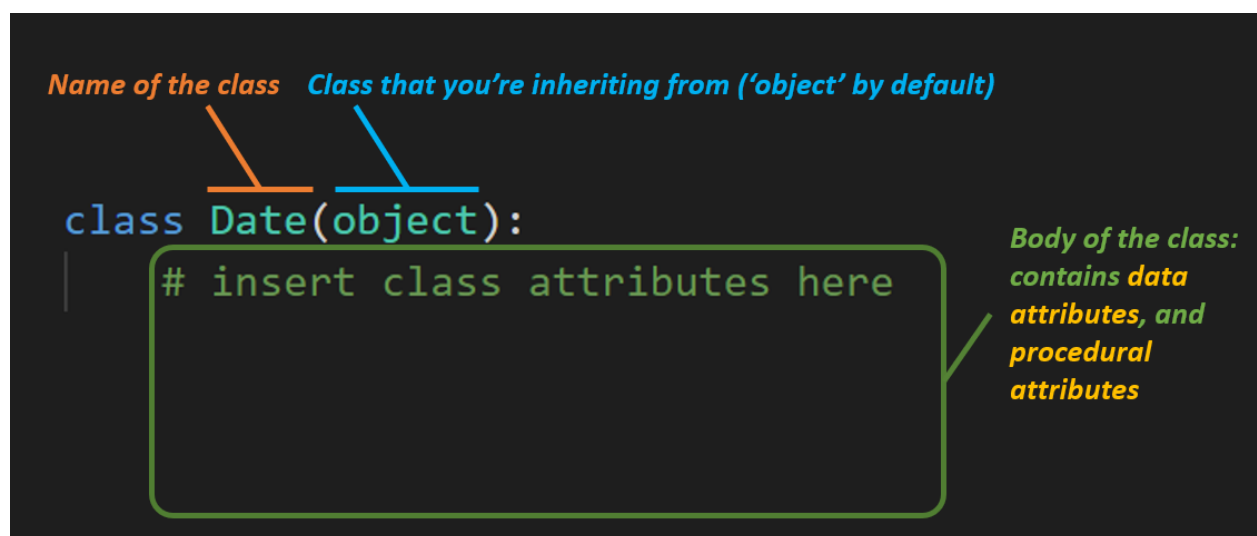
To read pickles, try the following: ``` with open('file_name.bin', 'rb') as file_rb: \n data = pickle.load(file_rb) ```

1.5: Object-Oriented Programming in Python

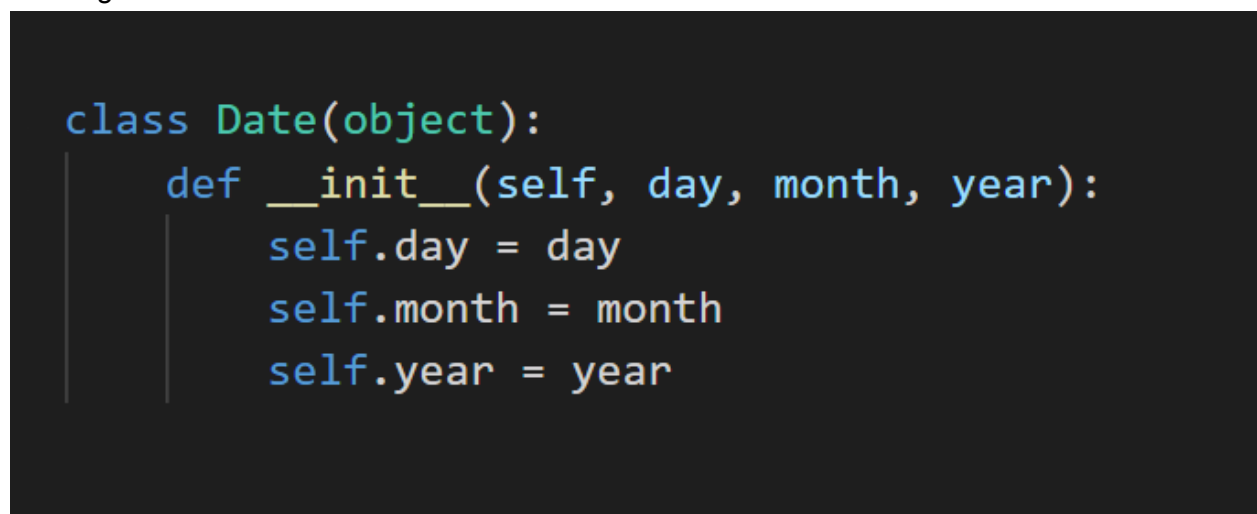
Learning Goals

- Apply object-oriented programming concepts to your Recipe app

What is Object-Oriented Programming? OOP involves abstracting data into objects to keep the code “dry” (don’t repeat yourself). “Everything in Python is an object, based on a class.” A class is like a template of an object. It is comprised of “data attributes” and “procedural attributes”. Data attributes are just stored values, e.g., ‘name’: ‘Bob’. Procedural attributes are methods that perform preconfigured functions.



Defining data attributes:



Inheritance:

Class to inherit properties from

```
class Height(object):  
    def __init__(self, feet, inches):  
        self.feet = feet  
        self.inches = inches  
  
    def __str__(self):  
        output = str(self.feet) + " feet, " +
```

```
class Person:  
    def walk():  
        print("Hello, I can walk!")
```

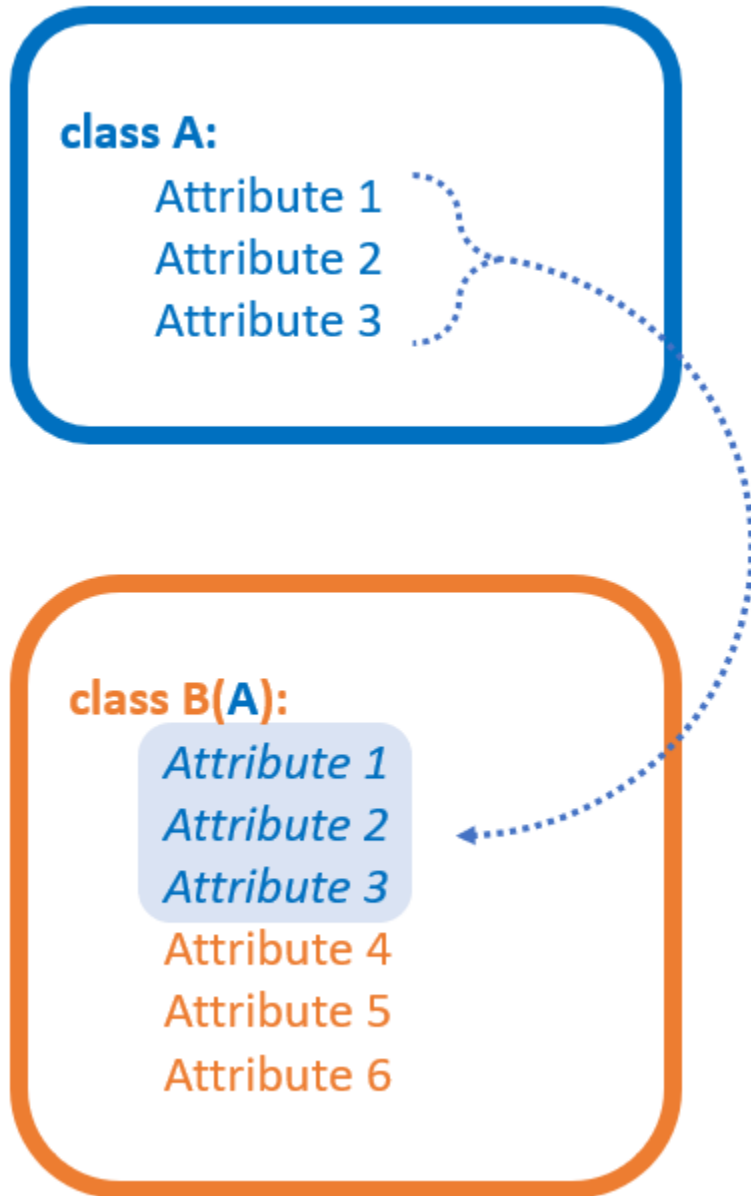
First, note how when building a class for the first time, you don't actually need to specify the `object` parameter yet (i.e., the class that's being inherited from). You can just enter `class` followed by the name of the class you're building (e.g., `class Height`).

Consider `Person` as your base class, and assume a new class called `Athlete`. You'll have `Athlete` inherit all of `Person`'s properties, while defining another method under it, called `run()`:

```
class Athlete(Person):  
    def run():  
        print("Hey, I can run too!")
```

The above text (directly from CareerFoundry) is important! This shows that you can define a class, then define another class taking the first class as an argument. Which will inherit all of the first class's attributes!

Like so:



1.6: Databases in Python

Learning Goals

- Create a MySQL database for your Recipe app

The MySQL database was a free download, with relatively easy setup. Though CareerFoundry introduced me to working with databases by diving into MongoDB (a non-relational database) I have started to warm up to SQL. The syntax is relatively natural, and the CRUD (Create = INSERT, Read = READ, Update = UPDATE, Delete = DELETE) operations are not brain-surgery to interact with.

I love spreadsheets, Excel is great, and SQL captures the charm of columns, rows, and cells when structuring data. Step aside, MongoDB. SQL is great.

Though, it is tricky to enter SQL syntax into Python... two languages at once.

```
(cf-python-base) C:\Users\Guy\Documents\GitHub\Python\Exercise_1.6>ipython recipe_mysql.py
=====
MAIN MENU

Choices:
1. Create a new recipe
2. Search for a recipe by ingredient
3. Update an existing recipe
4. Delete a recipe
5. View all recipes
Type "quit" to exit the program.
=====
Your choice: 2
=====
ALL INGREDIENTS

0. cereal
1. milk
2. a spoon
3. melons
4. syrup
5. batter
6. tootsie pops
=====
Which ingredient # would you like to search recipes for?: 6
=====
SEARCH RESULTS FOR RECIPES WITH: "TOOTSIE POPS"
-----
ID: 11
Name: Nachos
Ingredients: melons, syrup, batter, tootsie pops
Cooking Time: 10
Difficulty: Hard
-----
```

1.7: Object-Relational Mapping in Python

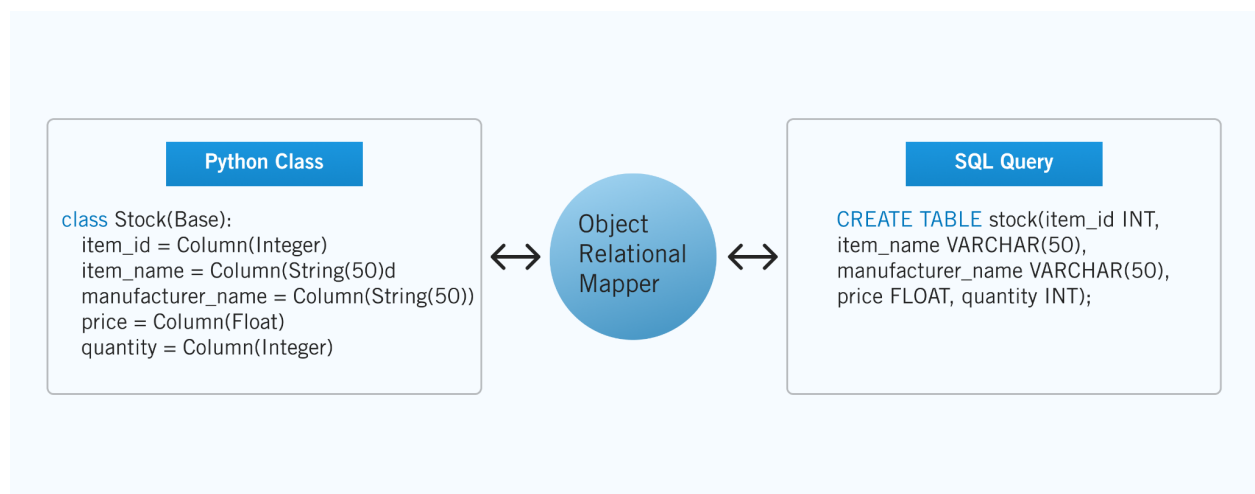
Learning Goals

- Interact with a database using an object-relational mapper
- Build your final command line Recipe application

Here's an example of an item being added to "Stock" with ORM:

```
new_item = Stock(  
    item_id = 1,  
    item_name = "Water",  
    manufacturer_name = "Aquafina",  
    price = 10,  
    quantity = 20  
)
```

```
session.add(new_item)  
session.commit()
```



Object Relational Mapping allows for interactions with data from a SQL database to be treated as Python objects. This means that instead of using Python, AND SQL syntax, you can just use Python.

2.1: Getting Started with Django

Learning Goals

- Explain MVT architecture and compare it with MVC
- Summarize Django's benefits and drawbacks
- Install and get started with Django

A key concept to grasp is the definition of MVT and how it differs from MVC. MVC is “Model, View, Controller” while MVT is “Model View Template”.

With MVC, Model is the Database. View is what renders and collects data in the browser. Controller is the logic between the model and the view.

With MVT, Model is the Database. View is the business logic that fetches from the database and gives to the frontend. Template is the user interface and renders things in the browser.

Pre-Requisite Checks

Python: version 3.8.7

```
(web-dev) C:\Users\Guy>python --version
Python 3.8.7
```

Activated Virtual Environment: named “web-dev”

```
(web-dev) C:\Users\Guy>C:\Users\Guy\Envs\web-dev\Scripts\activate.bat
(web-dev) C:\Users\Guy>if defined _OLD_VIRTUAL_PYTHONPATH (set "PYTHONPATH=" ) else (set "_OLD_VIRTUAL_PYTHONPATH=" )
```

Django installed in the environment:

```
(web-dev) C:\Users\Guy>py -m pip install Django
Collecting Django
  Downloading Django-4.2.2-py3-none-any.whl (8.0 MB)
----- 8.0/8.0 MB 2.2 MB/s eta 0:00:00
Collecting asgiref<4,>=3.6.0 (from Django)
  Downloading asgiref-3.7.2-py3-none-any.whl (24 kB)
Collecting sqlparse>=0.3.1 (from Django)
  Downloading sqlparse-0.4.4-py3-none-any.whl (41 kB)
----- 41.2/41.2 kB 998.2 kB/s eta 0:00:00
Collecting backports.zoneinfo (from Django)
  Downloading backports.zoneinfo-0.2.1-cp38-cp38-win_amd64.whl (38 kB)
Collecting tzdata (from Django)
  Downloading tzdata-2023.3-py2.py3-none-any.whl (341 kB)
----- 341.0/341.8 kB 2.1 MB/s eta 0:00:00
Collecting typing-extensions>=4 (from asgiref<4,>=3.6.0->Django)
  Downloading typing_extensions-4.6.3-py3-none-any.whl (31 kB)
Installing collected packages: tzdata, typing-extensions, sqlparse, backports.zoneinfo, asgiref, Django
Successfully installed Django-4.2.2 asgiref-3.7.2 backports.zoneinfo-0.2.1 sqlparse-0.4.4 typing-extensions-4.6.3 tzdata-2023.3

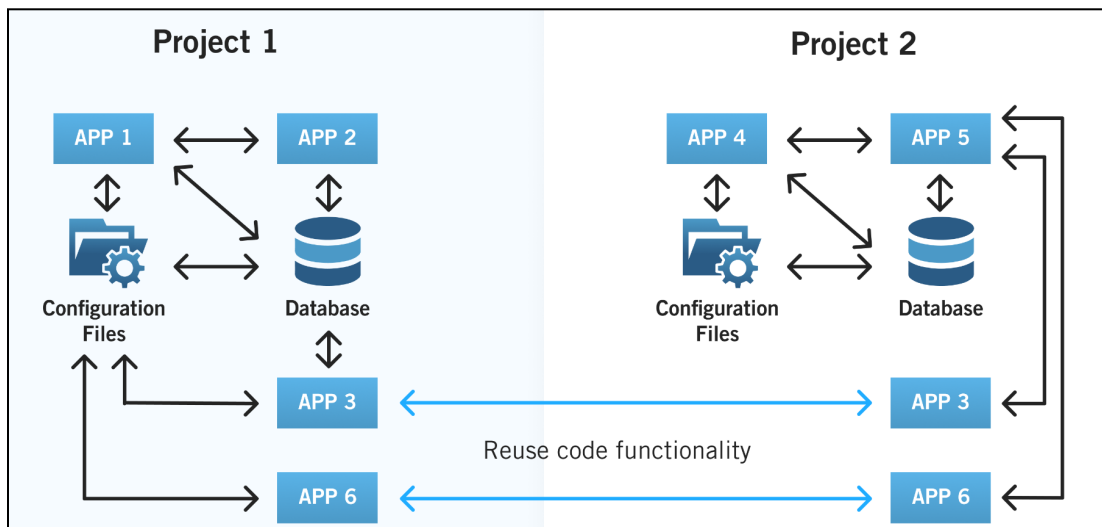
(web-dev) C:\Users\Guy>django-admin --version
4.2.2
```

2.2: Django Project Set Up

Learning Goals

- Describe the basic structure of a Django project
- Summarize the difference between Django projects and apps
- Create a Django project and run it locally
- Create a superuser in the admin interface of a Django web application

In Django a “Project” is the entire application structure. An “App” is a module with specific functionality. Apps are reusable pieces to keep things dry and use in multiple projects.



Creating a Django Project Step-by-Step

```
mkvirtualenv web-dev
```

```
web-dev\Scripts\activate.bat
```

```
pip install django
```

```
django-admin.exe startproject bookstore
```

Some possible commands that you can run via `manage.py` (or `django-admin`) are:

| COMMAND | ACTION |
|-----------------------------------------------------|----------------------------------------------|
| <code>check</code> | Checks the framework for common problems |
| <code>migrate</code> or <code>makemigrations</code> | Creates/Updates database |
| <code>runserver</code> | Runs a test server |
| <code>diffsettings</code> | Checks for differences from default settings |
| <code>sendtestemail</code> | Sends a test email |
| <code>startapp</code> | Creates an app |
| <code>test</code> | Runs tests |

Run Migrations:

```
...\> py manage.py migrate
```

Run Server:

```
py manage.py runserver
```

Create an App ("books" app):

```
...\> py manage.py startapp books
```

Create superuser:

```
python manage.py createsuperuser
```

2.3: Django Models

Learning Goals

- Discuss Django models, the “M” part of Django’s MVT architecture
- Create apps and models representing different parts of your web application
- Write and run automated tests

Think of database tables as Django “Models”.

“**Models** are Python objects that Django web applications use to access and manage data from the database. Django models define the structure of the data stored in the database, including field types, default values, the maximum size of the stored data, and so on.”

Step-by-Step

Create an App for each database table:

```
...\> py manage.py startapp books
```

In the `settings.py` file, look for the `INSTALLED_APPS` variable

Add each of the created apps to this list like so:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    #bookstore-related apps  
    'books',  
    'sales',  
    'salespersons',  
    'customers',  
]
```

Django Models: Defining, Registering, Migrating, and Running

Define the Django models for the application and then interact with them in the browser. It's a four-step process:

1. Defining the model: In this step, you'll specify the model (i.e., your table as a class).
2. Registering the model (class): You need to register your class in the `admin.py` file in your app. This will allow the class to be accessible from the Django admin.
3. Migration: After registration, you'll run commands on your terminal (using `Python manage.py`) to migrate the models. This will create the tables in the database.
4. Run server: Runs the server that you can access via your browser.

Step 1 Defining the Model

Navigate to the `customers` app in VSCode and open the `models.py` file.

Customer class code (customers/models.py file) will look as follows:

```
class Customer(models.Model):
    name= models.CharField(max_length=120)
    notes= models.TextField()

    def __str__(self):
        return str(self.name)
```

Step 2: Registering the Model (Class)

execute the following command in the `admin.py` file:

```
from .models import Customer
```

Next, to register, input the following (also in the `admin.py` file):

```
admin.site.register(Customer)
```

The `admin.py` file should now look as follows:

```
from .models import Customer
```

```
# Register your models here.
```

```
admin.site.register(Customer)
```

Step 3: Migration

Now, you need to migrate the class. Running migrations conveys any changes made to the model (class) across to the database.

NOTE!

Running migrations must be done each time a change is made to a model.

```
py manage.py makemigrations
```

```
py manage.py migrate
```

Step 4: Run Server

To run the server, execute the following command in your terminal:

```
py manage.py runserver
```

Now, head to Django admin
(`"http://127.0.0.1:8000/admin/"`) to see the Customers
section (Figure 8).

Writing Tests

To run tests with Django, go to each app's tests.py file

```
books/tests.py
```

You can now create your own class, such as MyTestClass, based on Django's TestCase, as follows:

```
class MyTestClass(TestCase):
```

```
    from .models import Book
```

```
    class BookModelTest(TestCase):
```

Your `setUpTestData` function will look like this:

```
    def setUpTestData():
        # Set up non-modified objects used by all test
methods
        Book.objects.create(name='Pride and Prejudice',
author_name='Jane Austen', genre='classic',
book_type='hardcover', price='23.71')
```

Then, in the same directory:

```
python manage.py test
```

Or, run tests just for one app:

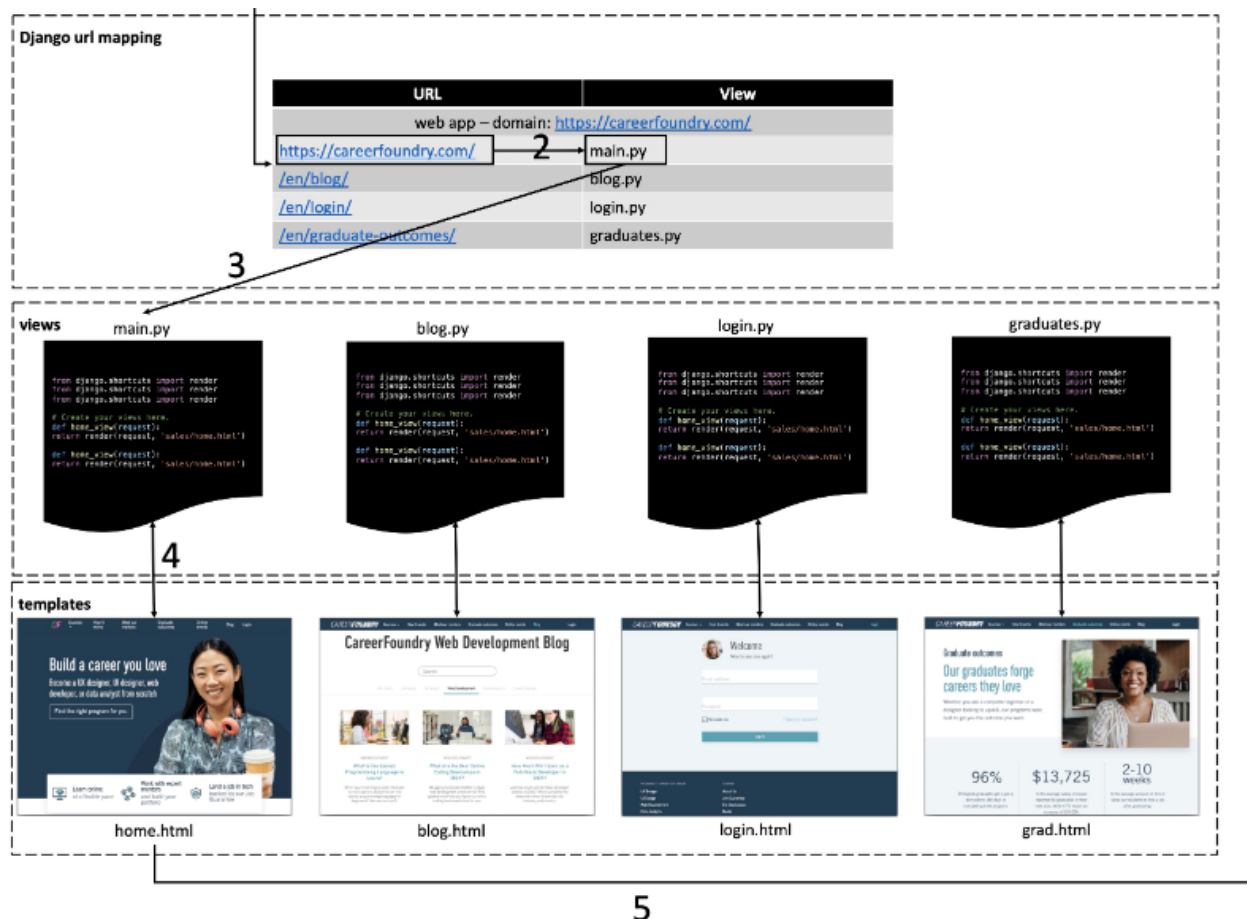
```
python manage.py test books
```

2.4: Django Views and Templates

Learning Goals

- Summarize the process of creating views, templates, and URLs
- Explain how the “V” and “T” parts of MVT architecture work
- Create a frontend page for your web application

“A view is the logic that Django runs when a user accesses a URL. Each view is represented as a Python function, or as a method of a Python class that accepts a request, runs Python code, and returns a response. The function can be quite basic, simply accepting a request and returning static information—such as an HTML web page, or JSON data—or the function could be as complex as a class that accepts a request, interacts with the database, accepts user input (sent as requests), and determines what will be returned to the user as a response.”

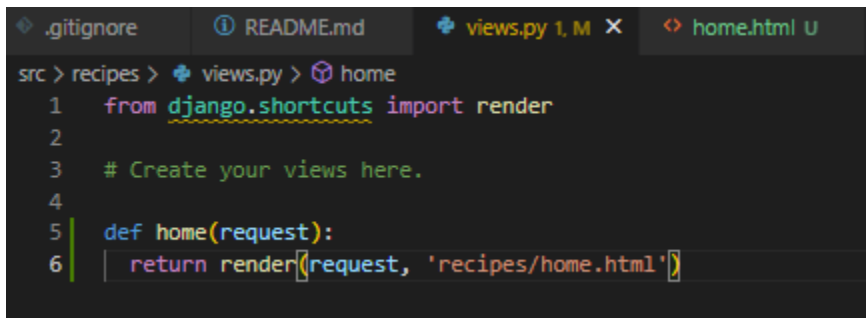


This custom look can be achieved by specifying the views, URLs, and templates in your app and registering them in your application. It's a four-step process:

1. Defining the view in the app/views.py file.
2. Creating the template(s) in the app/templates/ folder.
3. Mapping the URL to view in app/urls.py.
4. Registering the URL and view in project/urls.py.

Take careful note of the last three elements in the file structure. It's important to adhere to the following steps:

1. Create a folder named templates within your app (sales, in this instance).
2. Within the templates folder, create a new folder that is named the same as the app (i.e., sales).
3. Create the template pages as needed (e.g., home.html) within the new sales folder.
4. Specify the path to sales/home.html in the views.py file. (screenshot)



```
.gitignore  README.md  views.py 1, M  home.html U
src > recipes > views.py > home
1  from django.shortcuts import render
2
3  # Create your views here.
4
5  def home(request):
6      return render(request, 'recipes/home.html')
```

Nut shell of 2.4

Each “app” has templates and views. The “Template” defines what the user will see in the “View”, in this case a simple HTML file named “home.html”. It's important to maintain a valid “Django savvy file structure” for things to run smoothly. Then, be sure to include an <app>/urls.py, then specify the url in the <project>/urls.py. Run the server and go to the url, Django will generate the “View” linked to that url. Good stuff!

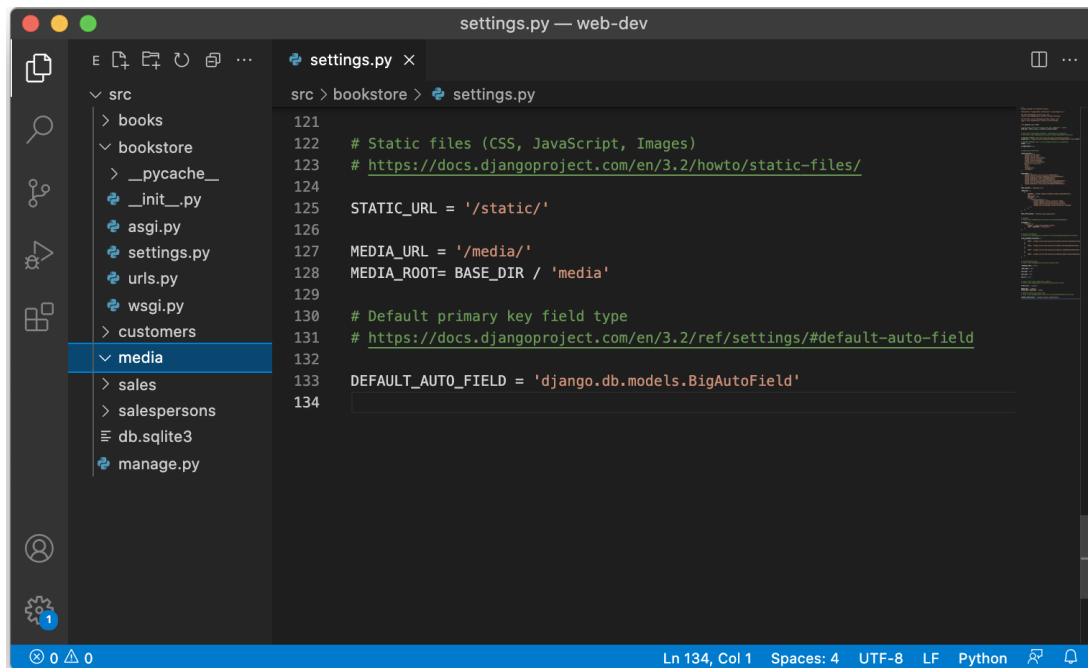
2.5: Django MVT Revisited

Learning Goals

- Add images to the model (database) and display them on the frontend of your application
- Create complex views with access to the model
- Display records with views and templates

The steps for adding images to a Django app are as follows:

1. Have a designated folder at the project (application) level where the images will be stored.
2. Specify the path to this image folder in the project's settings.py file.



3. Specify URL-View mapping in the project's urls.py file.

```
from django.conf import settings
from django.conf.urls.static import static
```

4. Add the pic attribute to model(s). Update the code in the models.py files of the individual apps to create new fields/columns in the database.

```
$pip install pillow
```

```

from django.db import models

# Create your models here.
class Customer(models.Model):
    name = models.CharField(max_length=120)
    notes = models.TextField()
    pic = models.ImageField(upload_to='customers', default='no_picture.jpg')

    def __str__(self):
        return str(self.name)

```

5. Provide a no-picture.jpg by default that the application can use in case an image isn't available for a certain entity. This step is optional.

Map the url in the “app”:

3. Map view to URL: Mapping the URL is done by creating a new file:

`books/urls.py`. This was already covered in Exercise 2.4. By following the relevant instructions there, you can update `books/urls.py` to the following code:

```

from django.urls import path
from .views import BookListView

app_name = 'books'

urlpatterns = [
    path('list/', BookListView.as_view(), name='list'),
]

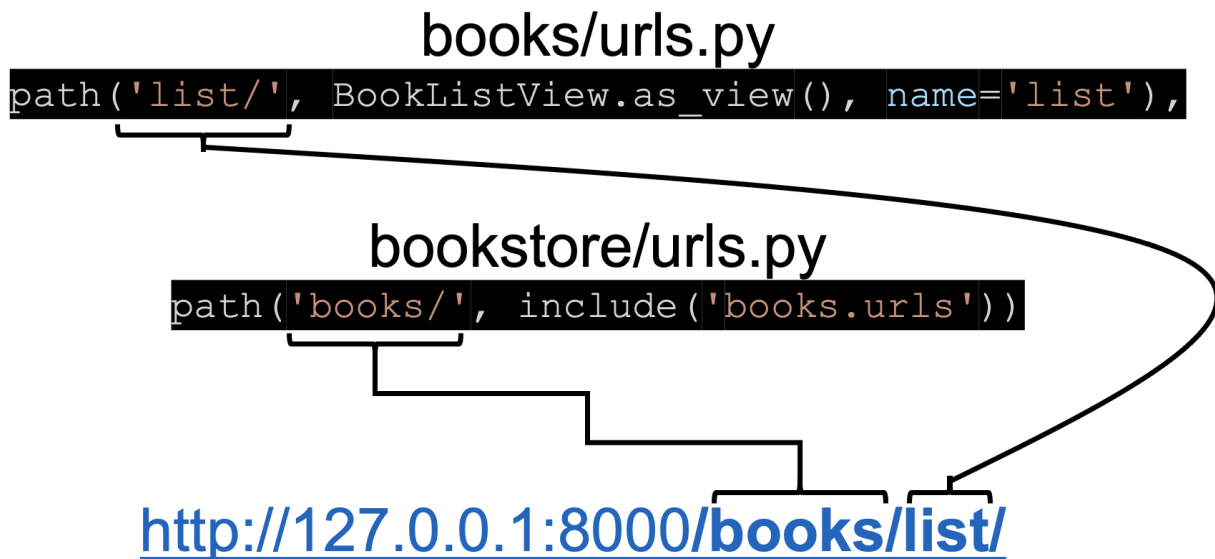
```

Then, register the url in the project:

4. Registering the View: Head over to the `bookstore/urls.py` file and register this file following the same process as in Exercise 2.4. Your updated `urlpatterns` variable in `bookstore/urls.py` should look as follows:

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

This is how the URL will look in the browser:



Okay, now to link between the list view and the detail view:

between the list page (that shows all books) and the details page of one book (specifically, the book that was clicked) This will be done in the `books/urls.py` file. Here, you add the following parameter to `urlpatterns` —this time to connect the URL to `BookDetailView`. Once again, since `DetailView` is class-based, you need to specify the method `as_view()`.

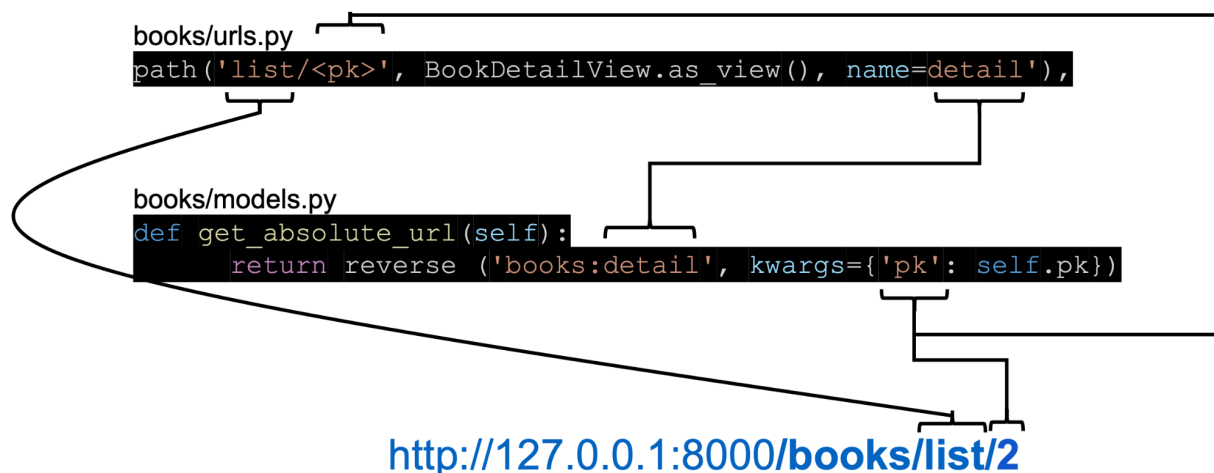
```
path('list/<pk>', BookDetailView.as_view(), name='detail'),
```

Making the book titles clickable and telling Django which book's details must be loaded involves the additional `<pk>` parameter. This parameter indicates the primary key of the object. You'll look into `<pk>` now. For this, go to `books/models.py` and define a function, `get_absolute_url()`, under the class `Book`. The `get_absolute_url()` function will take `<pk>` as the primary key and generate a URL:

```
def get_absolute_url(self):  
    return reverse ('books:detail', kwargs={'pk': self.pk})
```

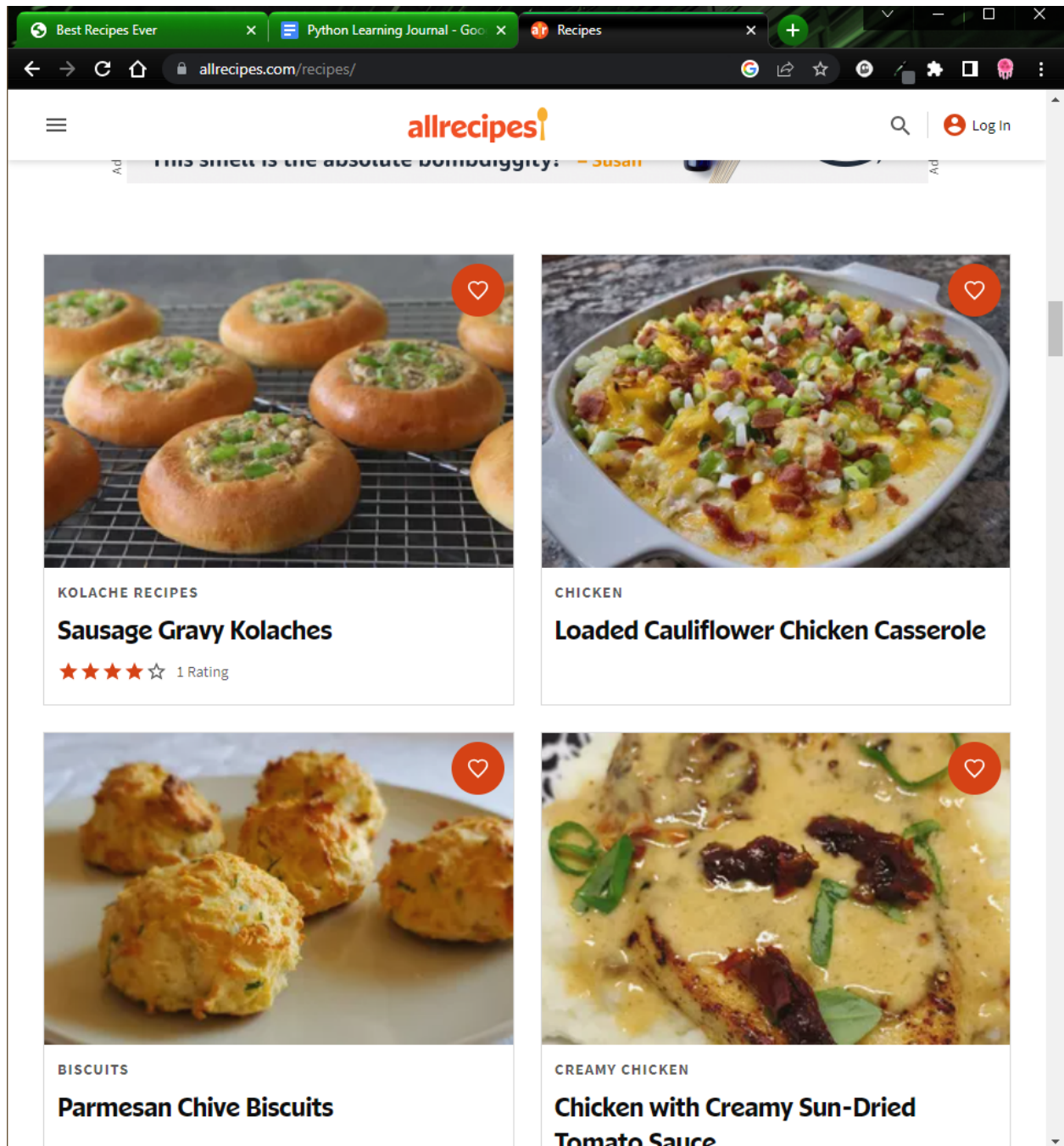
This needs a `reverse` function that can be imported from Django shortcuts:

```
from django.shortcuts import reverse
```



Finally, you need to update your `main.html` file, but instead of directly printing the `{{object.name}}` title, you need to add the following link: ` {{object.name}} `.

Frontend Inspirations



Allrecipes.com is the best recipe website in my opinion. It's super simple. It shows recipes as a list of cards with a prominent image, a "favorite heart", genre, title, and rating. Bam, those are the pieces of info you need.

2.6: User Authentication in Django

Learning Goals

- Learn how to create authentication for your web application
- Become familiar with using GET and POST methods
- Learn how to password protect your web application's views

Here's a great tip from CF:

TIP!

If you forget or lose your username or password, you can return to Django admin via two methods. Before you attempt them, make sure your virtual environment is active and in the `src` folder.

Number One:

If you still know your username, simply enter the command `python manage.py changepassword <user_name>` and your password will be reset.

Number Two:

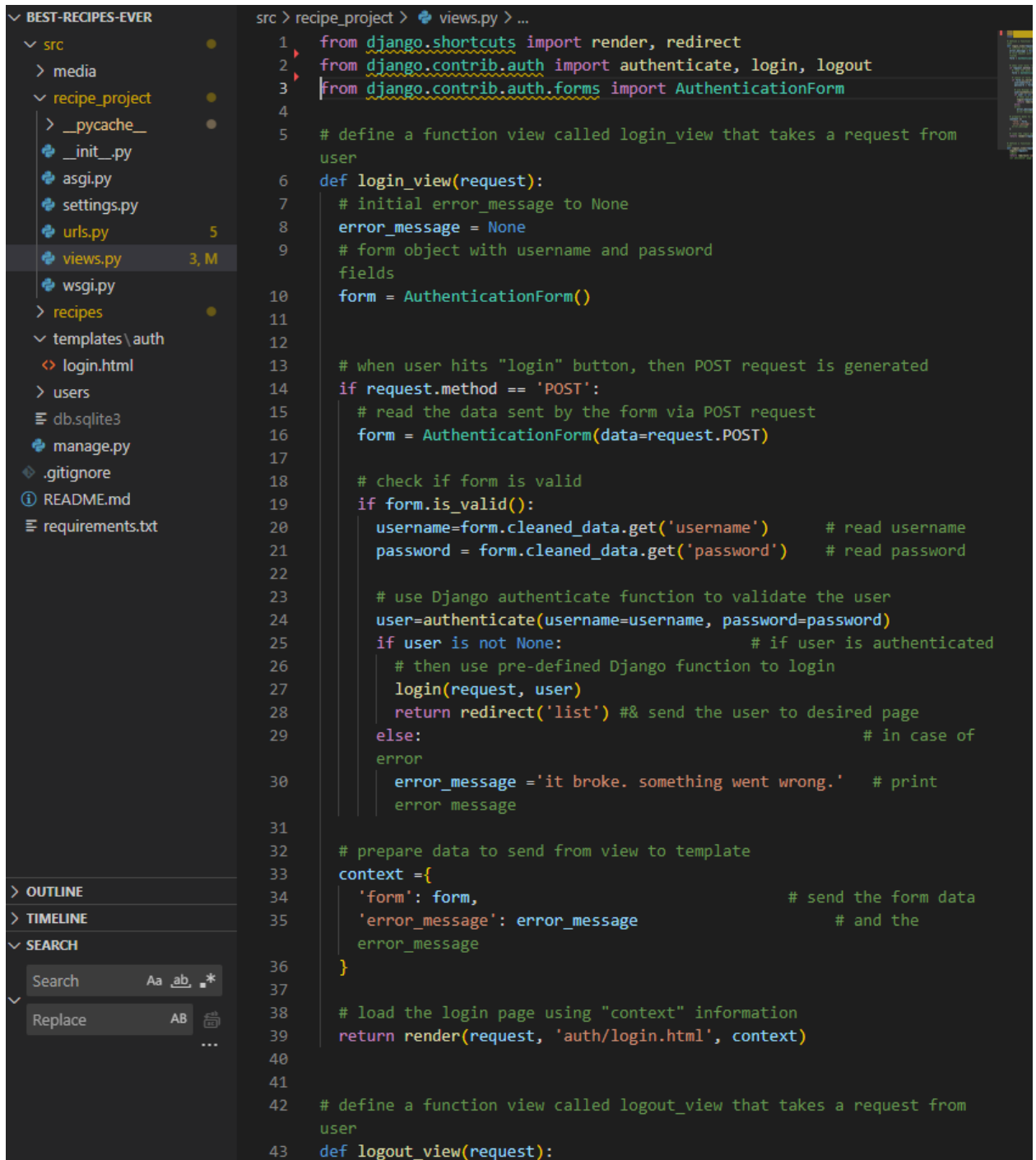
If you don't know your username, you'll have to create a new superuser using the command `python manage.py createsuperuser`. From there, you can recover your username by checking the admin dashboard.

To create authentication for a Django application, follow the established pattern:

1. Create a login.html “template” (at the project level make the “templates/auth” folders

```
src > templates > auth > login.html > html
1  {% load static %}
2  <!DOCTYPE html>
3  <html lang="en">
4
5  <head>
6      <title>BRE Login</title>
7      <meta charset="UTF-8">
8      <meta http-equiv="X-UA-Compatible" content="IE=edge">
9      <meta name="viewport" content="width=device-width, initial-scale=1.0">
10     <link rel="stylesheet" href="{% static 'recipes/css/draft.css' %}">
11 </head>
12
13 <body>
14     <div class="banner">
15         
17     </div>
18     <div class="container">
19         <h1>Best Recipes Ever</h1>
20         <h2>Login</h2>
21         {% comment %} check for error message {% endcomment %}
22         {% if error_message %}
23             <p style="color: red">{{error_message}}</p>
24         {% endif %}
25
26         {% comment %} print form {% endcomment %}
27         <form class="login-form" action="" method="POST">
28             {% comment %} add Django security token {% endcomment %}
29             {% csrf_token %}
30
31             {% comment %} print form received from view {% endcomment %}
32             {{form}}
33             {% comment %} add button {% endcomment %}
34             <br /><br />
35             <button class="button-big" type="submit">Login</button>
36         </form>
37     </div>
38
39 </body>
40
41 </html>
```

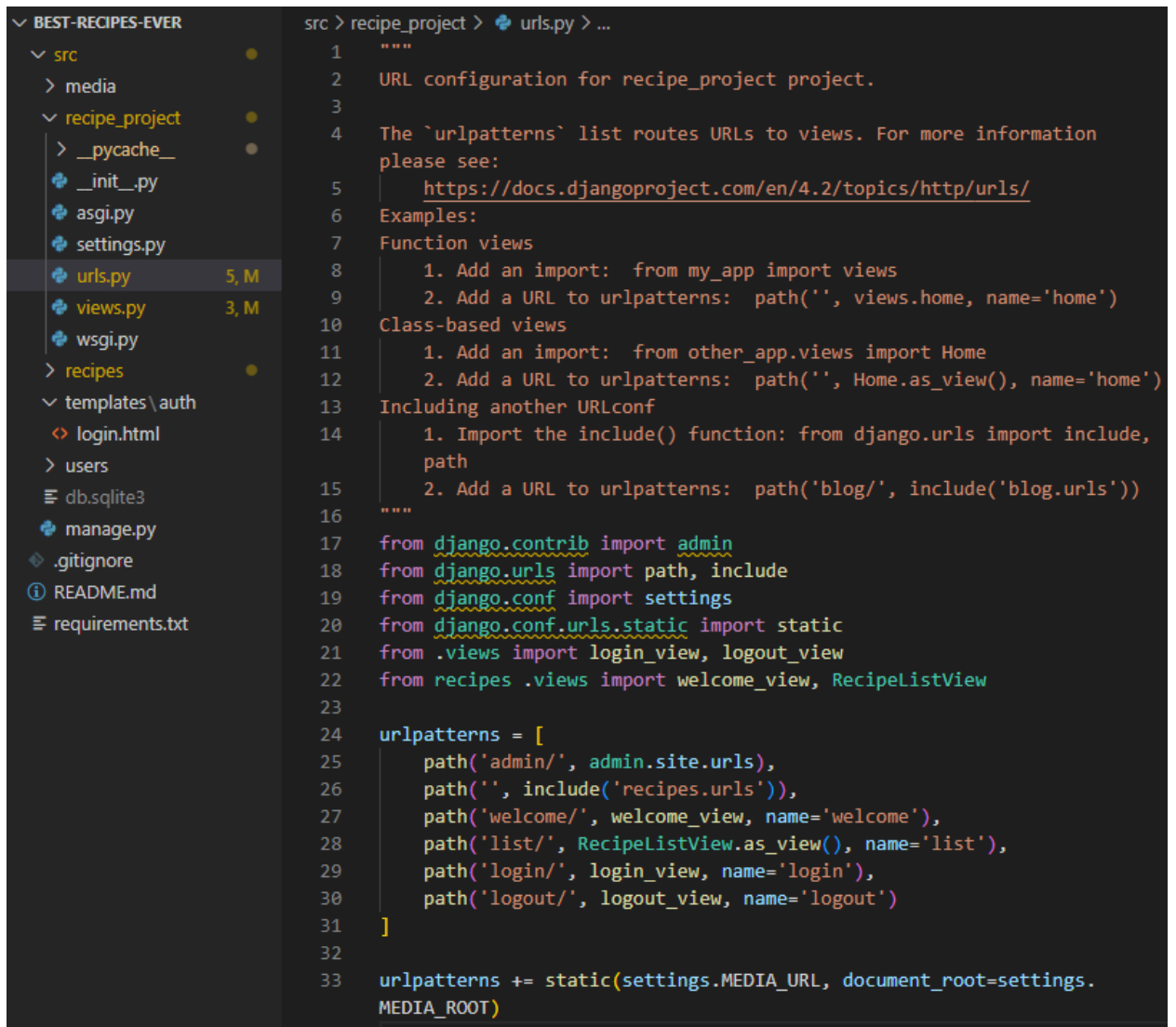

2. Define the “login view” and “logout_view” (project level “views.py”). In this case, these views are FBVs with Django's authentication packages imported like so:



The image shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows the project structure for 'BEST-RECIPES-EVER', including 'src', 'media', 'recipe_project', and 'templates'. The 'views.py' file is selected in the 'recipe_project' folder. The code editor shows the content of 'views.py', which defines two function-based views: 'login_view' and 'logout_view'. The 'login_view' function handles POST requests, validates the form, authenticates the user, and renders the 'login.html' template. The 'logout_view' function is defined but its implementation is not shown in the image.

```
src > recipe_project > views.py > ...
1  from django.shortcuts import render, redirect
2  from django.contrib.auth import authenticate, login, logout
3  from django.contrib.auth.forms import AuthenticationForm
4
5  # define a function view called login_view that takes a request from
   user
6  def login_view(request):
7      # initial error_message to None
8      error_message = None
9      # form object with username and password
      fields
10     form = AuthenticationForm()
11
12
13     # when user hits "login" button, then POST request is generated
14     if request.method == 'POST':
15         # read the data sent by the form via POST request
16         form = AuthenticationForm(data=request.POST)
17
18         # check if form is valid
19         if form.is_valid():
20             username=form.cleaned_data.get('username')      # read username
21             password = form.cleaned_data.get('password')    # read password
22
23             # use Django authenticate function to validate the user
24             user=authenticate(username=username, password=password)
25             if user is not None:                            # if user is authenticated
26                 # then use pre-defined Django function to login
27                 login(request, user)
28                 return redirect('list') #& send the user to desired page
29             else:                                           # in case of
30                 error_message = 'it broke. something went wrong.' # print
31                 error_message
32
33     # prepare data to send from view to template
34     context ={
35         'form': form,                                     # send the form data
36         'error_message': error_message                   # and the
37         error_message
38     }
39
40     # load the login page using "context" information
41     return render(request, 'auth/login.html', context)
42
43 # define a function view called logout_view that takes a request from
   user
44 def logout_view(request):
```

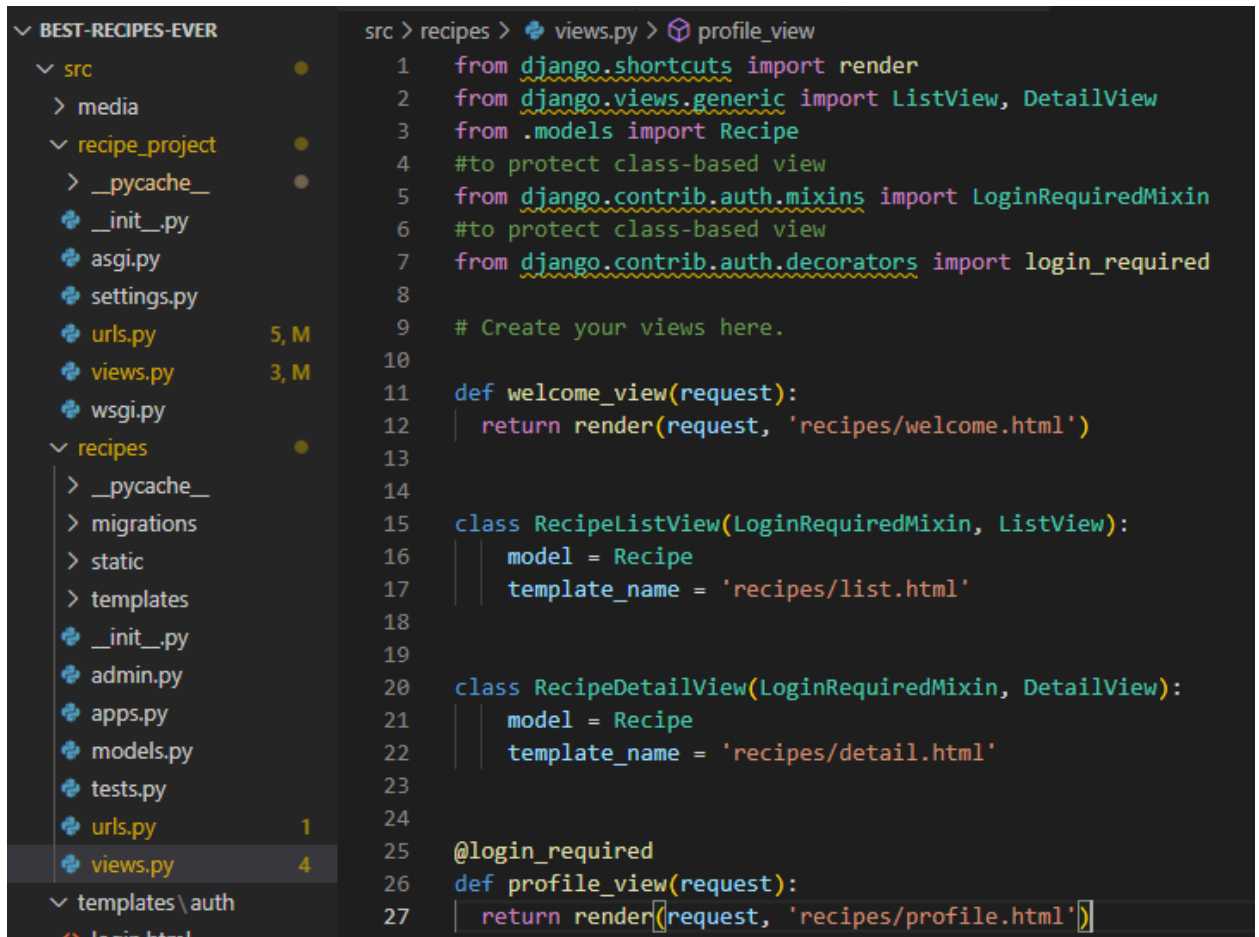
3. Okay, now hook up the URLs to these views in the project-level “views.py” file like so:



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure for 'BEST-RECIPES-EVER' with a 'src' directory containing 'media', 'recipe_project', and 'recipes'. The 'recipe_project' directory contains 'urls.py' (5, M) and 'views.py' (3, M). The 'urls.py' file is selected, and its content is displayed in the code editor. The code defines URL patterns for the project, including paths for 'admin/', 'welcome/', 'list/', 'login/', and 'logout/'.

```
src > recipe_project > urls.py > ...
1  """
2  URL configuration for recipe_project project.
3
4  The `urlpatterns` list routes URLs to views. For more information
   please see:
5  |     https://docs.djangoproject.com/en/4.2/topics/http/urls/
6  | Examples:
7  | Function views
8  |     1. Add an import: from my_app import views
9  |     2. Add a URL to urlpatterns: path('', views.home, name='home')
10 | Class-based views
11 |     1. Add an import: from other_app.views import Home
12 |     2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
13 | Including another URLconf
14 |     1. Import the include() function: from django.urls import include,
   |     path
15 |     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
16 | """
17 from django.contrib import admin
18 from django.urls import path, include
19 from django.conf import settings
20 from django.conf.urls.static import static
21 from .views import login_view, logout_view
22 from recipes.views import welcome_view, RecipeListView
23
24 urlpatterns = [
25     path('admin/', admin.site.urls),
26     path('', include('recipes.urls')),
27     path('welcome/', welcome_view, name='welcome'),
28     path('list/', RecipeListView.as_view(), name='list'),
29     path('login/', login_view, name='login'),
30     path('logout/', logout_view, name='logout')
31 ]
32
33 urlpatterns += static(settings.MEDIA_URL, document_root=settings.
   MEDIA_ROOT)
```

- Now, go into the (app level) views that you want to require a login to see. For CBVs, import Django's `LoginRequiredMixin` package. For FBVs, import Django's `login_required` package. Add the mixin as the first argument to the CBV class declaration. Or, for an FBV, write `@login_required` just above the defined view function.



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'BEST-RECIPES-EVER' with a 'src' directory containing 'media', 'recipe_project', and 'recipes'. The 'recipe_project' directory contains '__pycache__', '__init__.py', 'asgi.py', 'settings.py', 'urls.py', 'views.py', and 'wsgi.py'. The 'recipes' directory contains '__pycache__', 'migrations', 'static', 'templates', '__init__.py', 'admin.py', 'apps.py', 'models.py', 'tests.py', 'urls.py', and 'views.py'. The 'views.py' file in the 'recipes' directory is selected. The code editor shows the following code:

```
src > recipes > views.py > profile_view
1  from django.shortcuts import render
2  from django.views.generic import ListView, DetailView
3  from .models import Recipe
4  #to protect class-based view
5  from django.contrib.auth.mixins import LoginRequiredMixin
6  #to protect class-based view
7  from django.contrib.auth.decorators import login_required
8
9  # Create your views here.
10
11 def welcome_view(request):
12     return render(request, 'recipes/welcome.html')
13
14
15 class RecipeListView(LoginRequiredMixin, ListView):
16     model = Recipe
17     template_name = 'recipes/list.html'
18
19
20 class RecipeDetailView(LoginRequiredMixin, DetailView):
21     model = Recipe
22     template_name = 'recipes/detail.html'
23
24
25 @login_required
26 def profile_view(request):
27     return render(request, 'recipes/profile.html')
```

And there you go. Now you've got a login form that authenticates with the Django user's credentials, and you can restrict views based on user login.

2.7: Data Analysis and Visualization in Django

Learning Goals

- Work on elements of two-way communication like creating forms, getting input from the user, and creating buttons
- Implement search and visualization (reports/charts) features
- Use QuerySet API, DataFrames (using pandas) and plotting libraries (using matplotlib)

How do you make forms to capture user input?

1. Make a “forms.py” file in the desired app.
2. Import “forms” from django
3. Define the form as a class with “forms.Form” as an argument

```
src > recipes > forms.py > ...
1  from django import forms
2
3  CHART_CHOICES = ( # specify choices as a tuple
4      ('#1', 'Bar chart'), # when user selects "Bar chart", it is stored as "#1"
5      ('#2', 'Pie chart'),
6      ('#3', 'Line chart')
7  )
8
9  # define class-based Form imported from Django forms
10
11
12  class RecipeSearchForm(forms.Form):
13      recipe_name = forms.CharField(max_length=120)
14      chart_type = forms.ChoiceField(choices=CHART_CHOICES)
15
```

4. Now, import and plug the form in to a view in the app’s “views.py”

```
@login_required
def records_view(request):
    form = RecipeSearchForm(request.POST or None)
    recipes_df = None # initialize (pandas) dataframe

    if request.method == 'POST':
        recipe_name = request.POST.get('recipe_name')
        chart_type = request.POST.get('chart_type')
        print(recipe_name, chart_type)

        print('Exploring querysets:')
        print('Case 1: Output of Recipe.objects.all()')
        qs = Recipe.objects.all()
        print(qs)

        print('Case 2: Output of Recipe.objects.filter(name=recipe_name)')
        qs = Recipe.objects.filter(name=recipe_name)
        print(qs)

        print('Case 3: Output of qs.values()')
        print(qs.values())

        print('Case 4: Output of qs.values_list()')
        print(qs.values_list())

        print('Case 5: Output of Recipe.objects.get(id=1)')
        obj = Recipe.objects.get(id=1)
        print(obj)

    # pack up data to be sent to template in the context dictionary
    context = {
        'form': form,
    }
    # load the recipes/record.html page using the data that was just prepared
    return render(request, 'recipes/records.html', context)
```

5. Be sure to render the desired template with a “context” (passing form values)

2.8: Deploying a Django Application

Learning Goals

- Enhance the user experience and the look and feel of your web application using CSS and JS
- Deploy your Django web application on a web server
- Curate project deliverables for your portfolio

3.2. Configure `settings.py`:

Bring your ‘settings.py’ file back into the `bookstore/bookstore/` folder (remember, you removed it prior to uploading the project to GitHub due to its sensitive information). Open ‘settings.py’ and copy the following configuration to the bottom of the file:

```
# Heroku: Update database configuration from
$DATABASE_URL.
import dj_database_url
db_from_env = dj_database_url.config(conn_max_age=500)
DATABASES['default'].update(db_from_env)
```

4. Static Files:

Serving files from the Django *development* web server is okay during development, but it’s not recommended to continue doing so in production. It’s preferred to serve files from a Content Delivery Network (CDN) or the web server. To make it easy to host static files separately from the Django web application, Django provides the `collectstatic` tool to collect these files for

deployment. Django templates refer to the hosting location of the static files relative to a settings variable (`STATIC_URL`), so this can be changed if the static files are moved to another host/server. The relevant setting variables are:

`STATIC_URL`: This is the base URL location from which static files will be served.

`STATIC_ROOT`: This is the absolute path to a directory where Django's `collectstatic` tool will gather any static files referenced in your templates.

`STATICFILES_DIRS`: This lists additional directories that Django's `collectstatic` tool should search for static files.

Open the `bookstore/settings.py` file and head over to the section where static file (CSS, JS, images) information is shown.

```
# Static files (CSS, JavaScript, Images)
#
https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC_URL = '/static/'
STATICFILES_DIRS=[
    BASE_DIR / 'static'
]
```

Add the `STATIC_ROOT` path after these parameters and save the file.

Afterwards, this section should look as follows:

```
# Static files (CSS, JavaScript, Images)
```

```
#  
https://docs.djangoproject.com/en/3.2/howto/static-files/  
  
STATIC_URL = '/static/'  
STATICFILES_DIRS=[  
    BASE_DIR / 'static'  
]  
# The absolute path to the directory where collectstatic  
# will collect static files for deployment.  
STATIC_ROOT = BASE_DIR / 'staticfiles'
```

5. WhiteNoise: Continuing with serving static files, you'll need the **WhiteNoise** package. WhiteNoise allows Django web applications to serve their own static files. WhiteNoise is recommended by Heroku for use with Gunicorn in production.

5.1. Install WhiteNoise:

```
pip install whitenoise
```

5.2. Update `settings.py`:

Open the `bookstore/settings.py` file, find the `MIDDLEWARE` variable, and add the WhiteNoise information near the top of the list, after the **SecurityMiddleware** line:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'whitenoise.middleware.WhiteNoiseMiddleware',  
  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',
```

```
'django.contrib.auth.middleware.AuthenticationMiddleware',  
,  
'django.contrib.messages.middleware.MessageMiddleware',  
  
'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

VERY HELPFUL TIP: To run console commands on the Heroku server, :

```
`` heroku bash -a <app name> ``
```

And of course `` git push heroku main `` to push the repository to the Heroku deployment