Implementation Documentation:

This document intends to thoroughly explain the implementation of the important modules and functions of the system.

Form Modules:

The main thing we want to do on our website is create a nice way to add and edit blog posts. With forms we will have absolute power over our interface – we can do almost anything we can imagine!

AVI Module Forms:

We use Django built in form creator to create a basic form which will then have to pass our required attributes(which invoke database usage- to be discussed later) to it - ‘from Django import forms’.

### The Django [Form](https://docs.djangoproject.com/en/2.1/ref/forms/api/#django.forms.Form) class[¶](https://docs.djangoproject.com/en/2.1/topics/forms/#the-django-form-class)

At the heart of this system of components is Django’s [**Form**](https://docs.djangoproject.com/en/2.1/ref/forms/api/#django.forms.Form) class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a [**Form**](https://docs.djangoproject.com/en/2.1/ref/forms/api/#django.forms.Form) class describes a form and determines how it works and appears.

Django’s role in forms[**¶**](https://docs.djangoproject.com/en/2.1/topics/forms/#django-s-role-in-forms)

Handling forms is a complex business. Consider Django’s admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.

Django’s form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

* preparing and restructuring data to make it ready for rendering
* creating HTML forms for the data
* receiving and processing submitted forms and data from the client

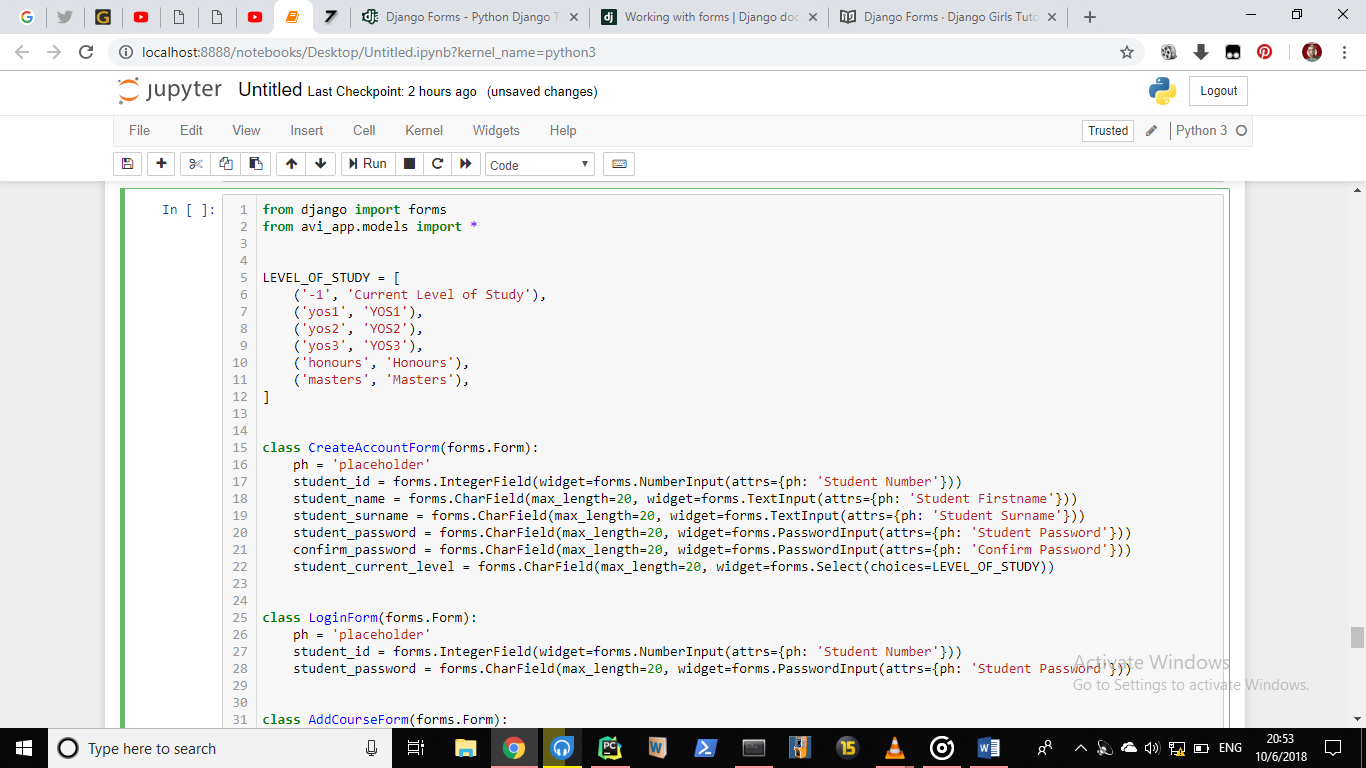
It is *possible* to write code that does all of this manually, but Django can take care of it all for you.

------------------------------------------------------------------------------------------------------------------------------------------

We link the form attributes to the models (database tables) pre-defined under the models python file of the app. This allows user input options and verification of requests inputs.

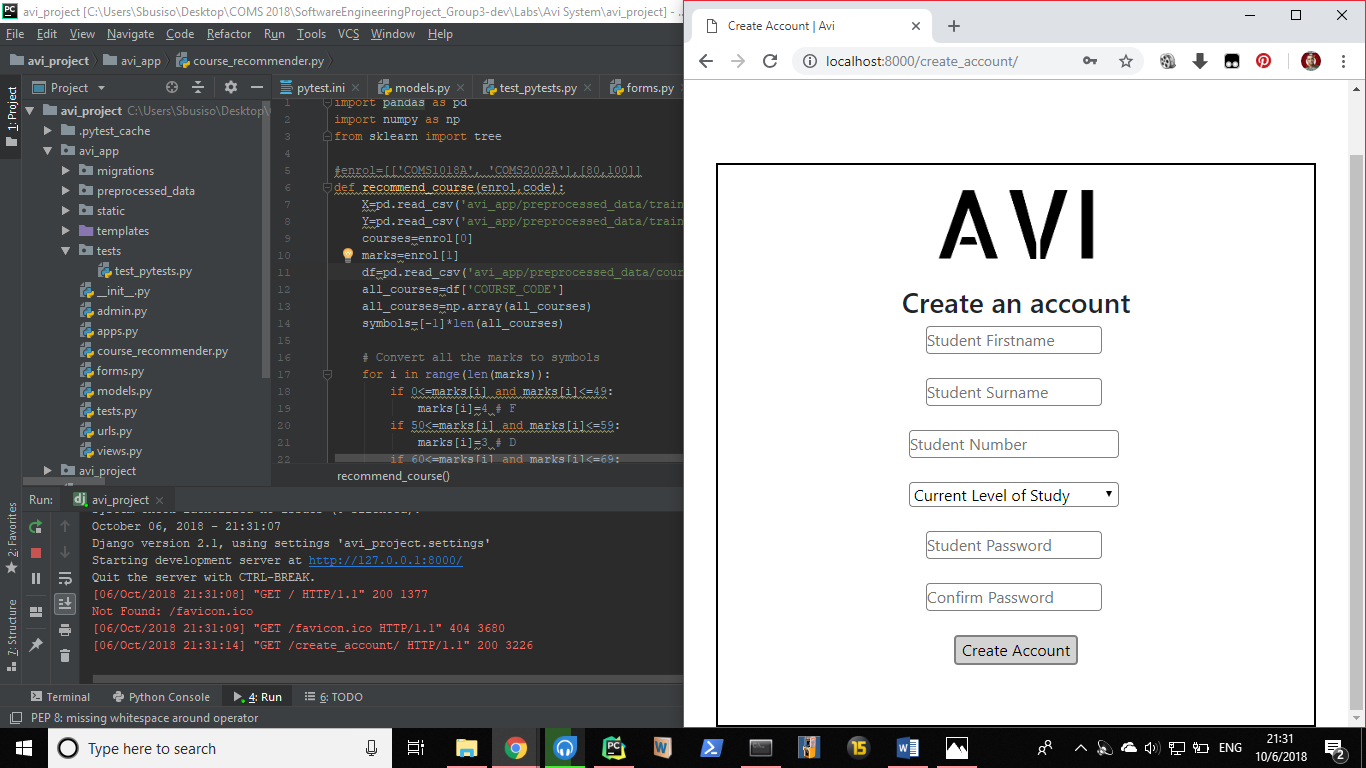
Module Explanation:

The Create User Form: Extracted from “avi\_app/forms.py”



The class ’CreateAccountForm(form.Form)’ instantiates the Student Class attributes therefore enabling them to be used in the form. We specify the type of input we can expect for each attribute which enables us to check whether the user input is valid. This class is used in incorporation with html code and CSS for the createaccount.html file.

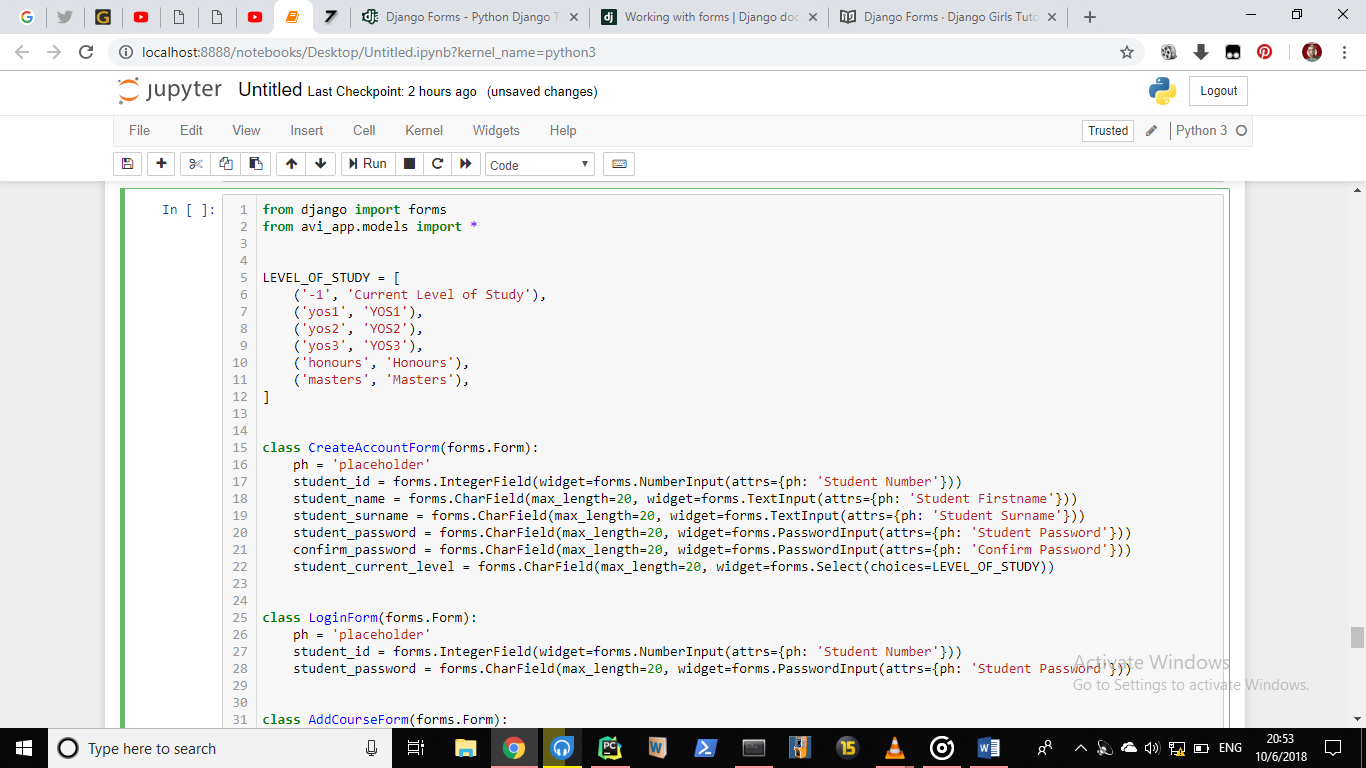
This form allows users to register to use the website app. This is what final form looks like:



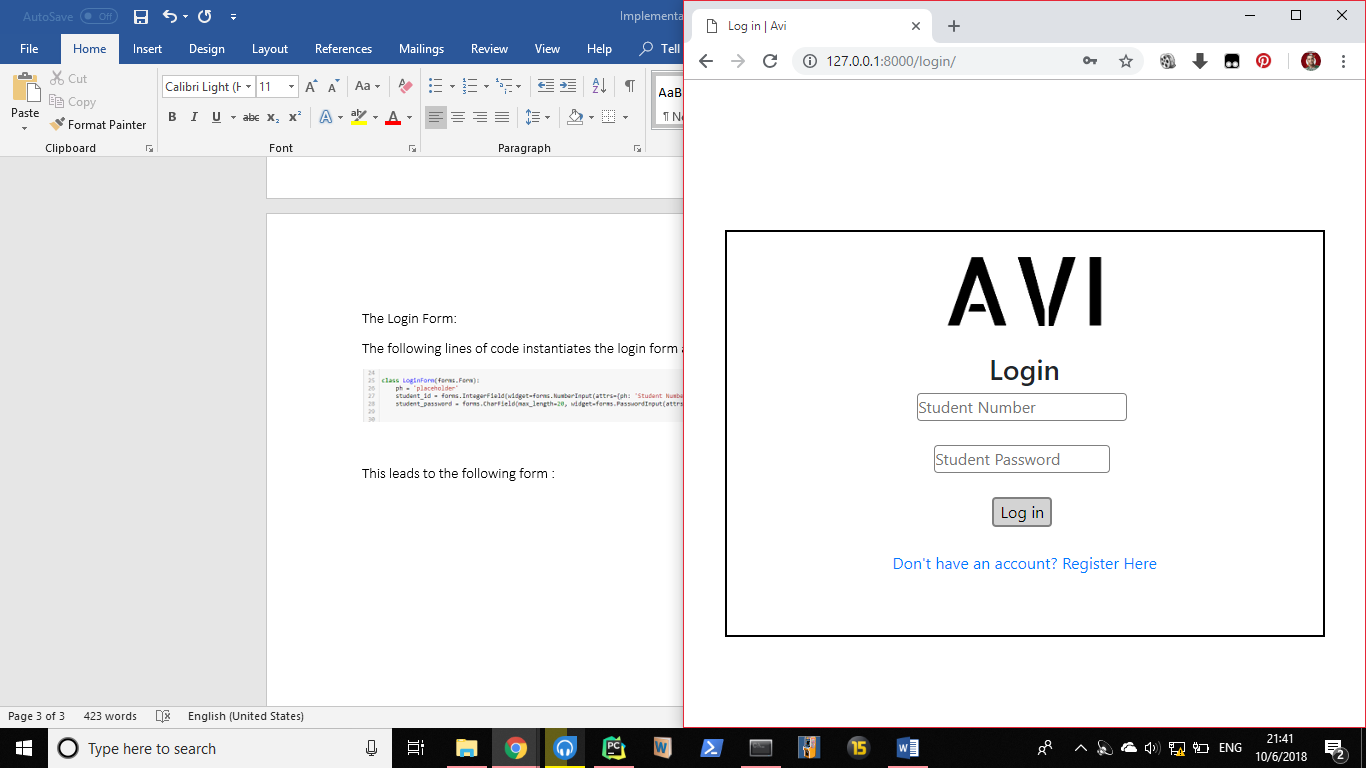
------------------------------------------------------------------------------------------------------------------------------------------

The Login Form : Extracted from “avi\_app/forms.py”

The following lines of code instantiates the login form attributes. This form allows user to be able to login to the website app.



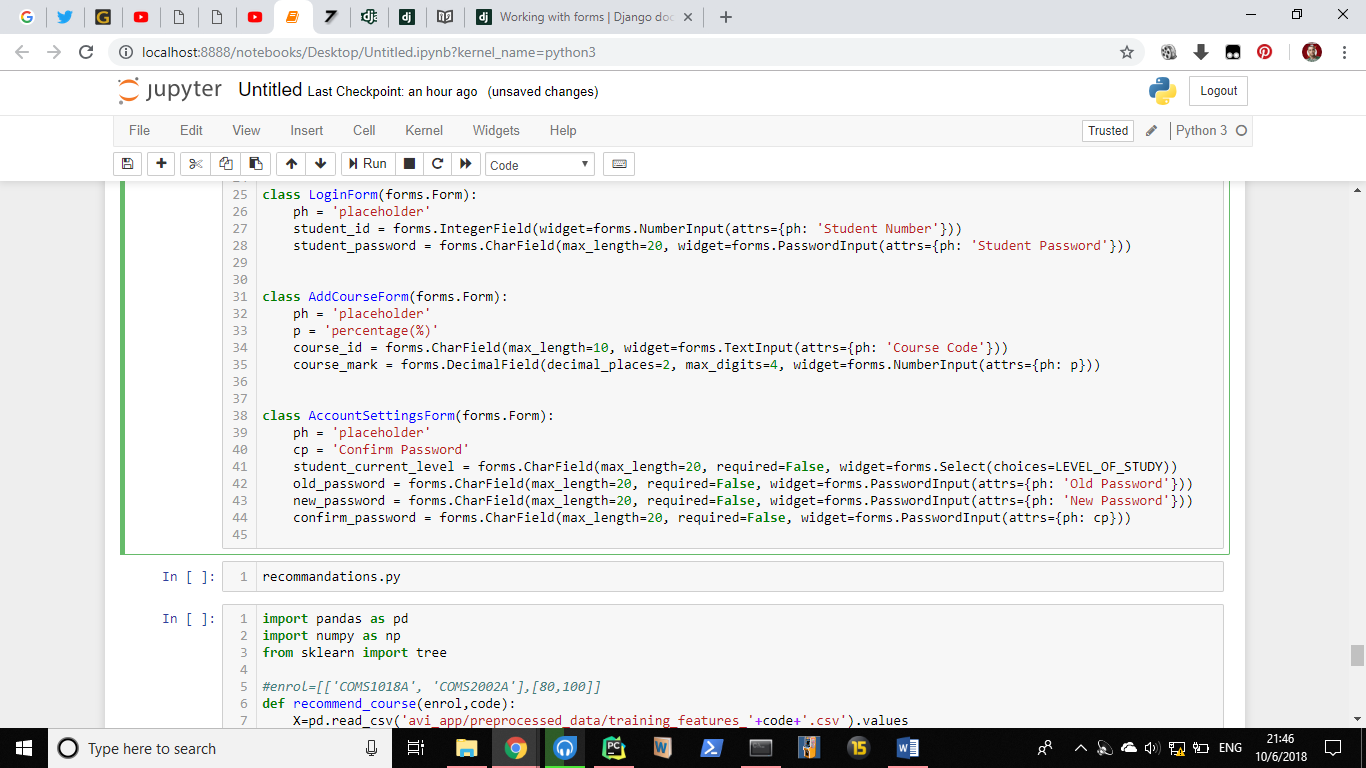
This along with the html and CSS code leads to the following form being created:



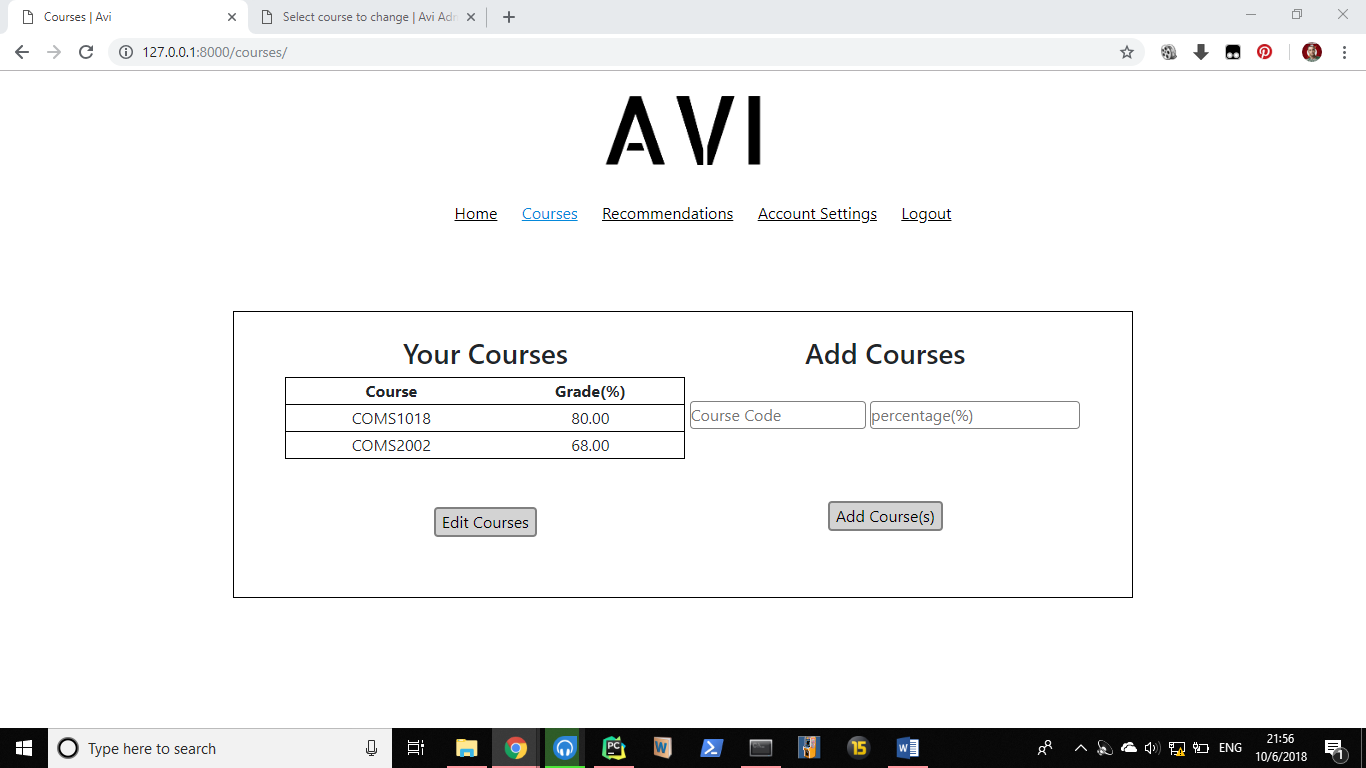
------------------------------------------------------------------------------------------------------------------------------------------

Add Courses Form : Extracted from “avi\_app/forms.py”

This form allows users to be add courses to their profile.

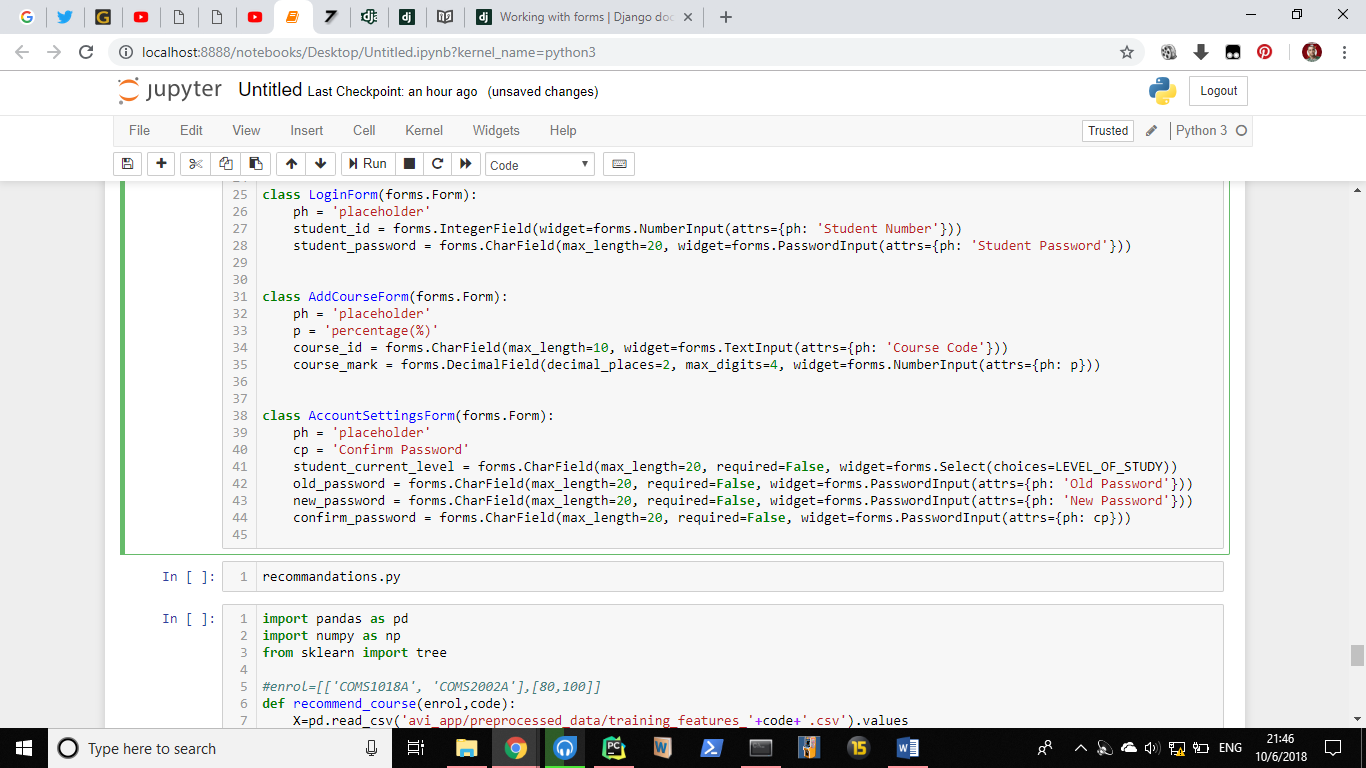


The above code along with the html and CSS code leads to the following form being created:

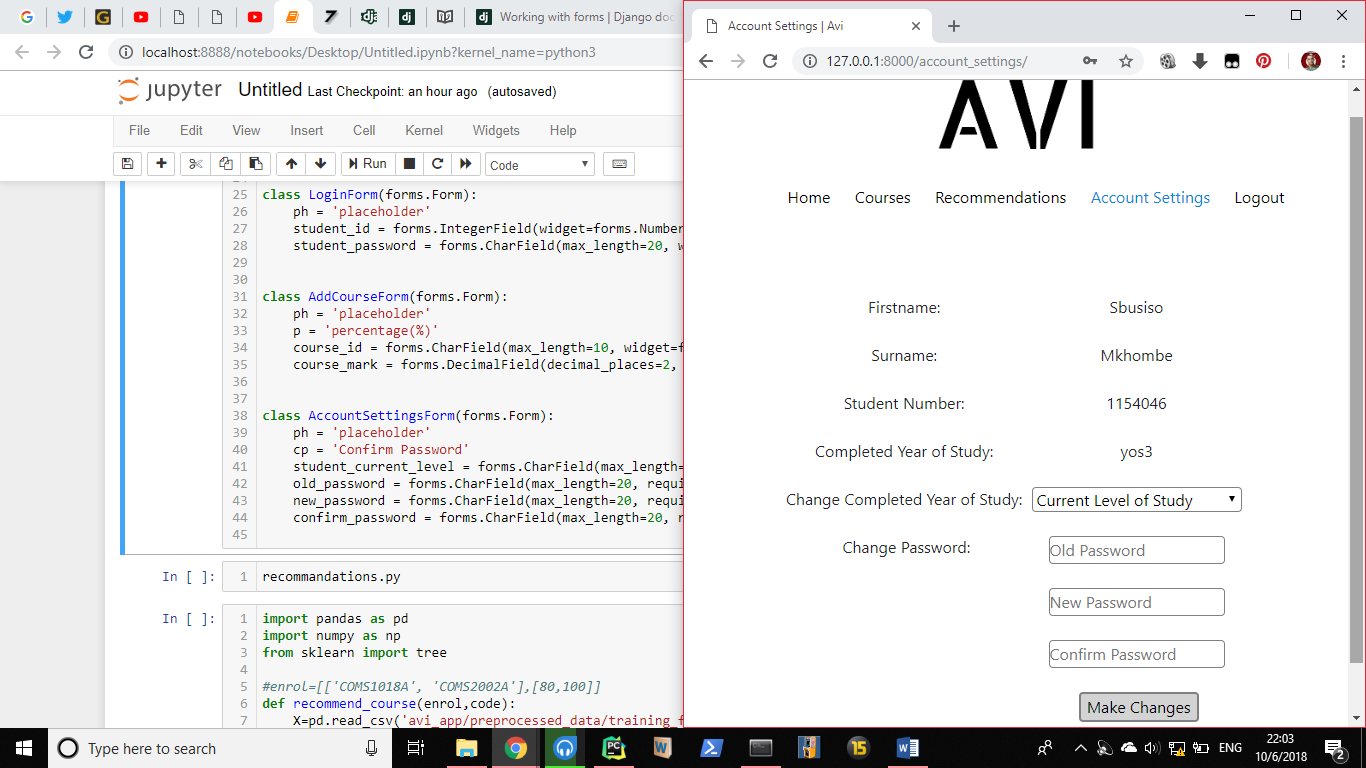


The Account Settings Form: Extracted from “avi\_app/forms.py”

The following lines of code instantiates foreign key attributes. This form allows user to be able to edit their personal details in the website app.



The above code along with the html and CSS code leads to the following form being created:



**Models:**

Also known as model attributes. A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you’re storing. Generally, each model maps to a single database table.

## Using models[¶](https://docs.djangoproject.com/en/2.1/topics/db/models/#using-models)

Once you have defined your models, you need to tell Django you’re going to use those models. Do this by editing your settings file and changing the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-INSTALLED_APPS) setting to add the name of the module/ app that contains your **models.py**.

## Fields[¶](https://docs.djangoproject.com/en/2.1/topics/db/models/#fields)

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attribute. Examples of fields in our Student Model (Below) would be student\_name, student\_surname and student\_id.

### **Field options**[**¶**](https://docs.djangoproject.com/en/2.1/topics/db/models/#field-options)

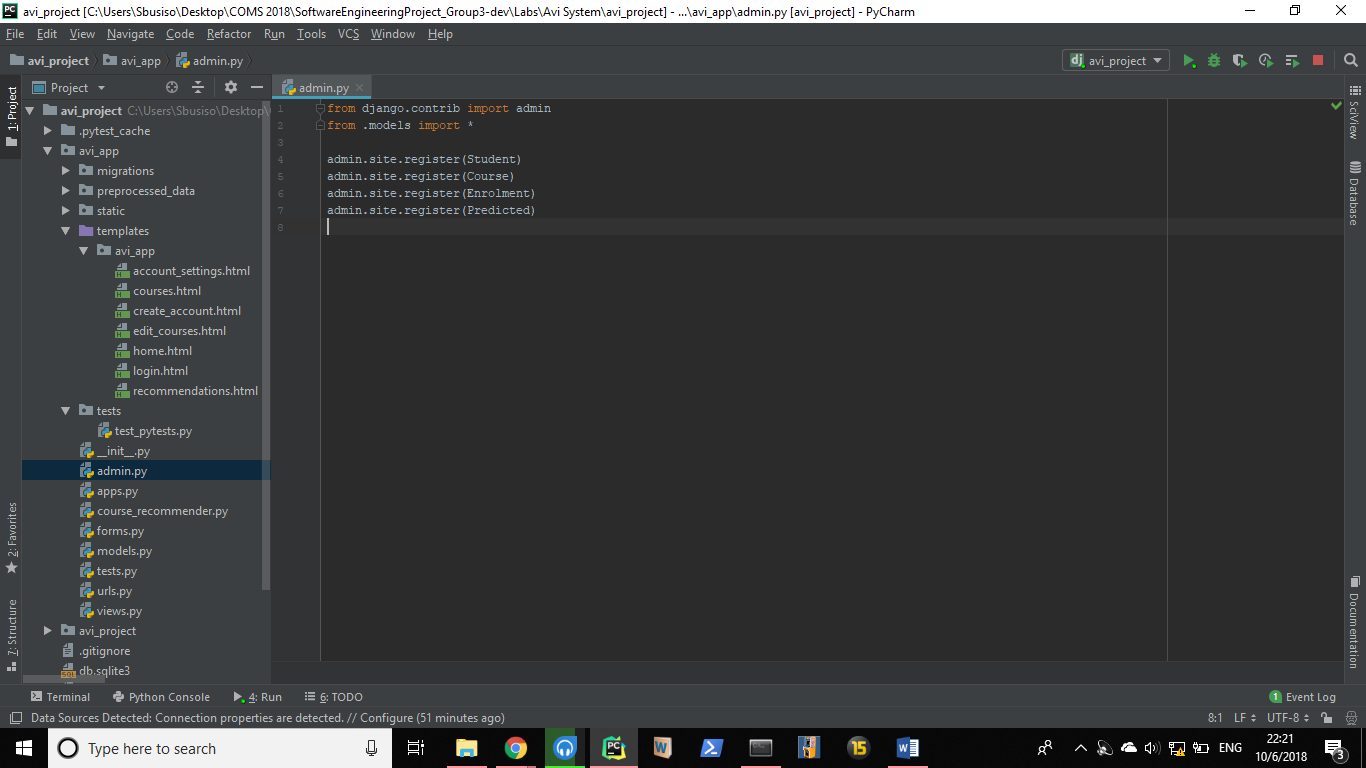
Each field takes a certain set of field-specific arguments (documented in the [model field reference](https://docs.djangoproject.com/en/2.1/ref/models/fields/#model-field-types)). For example, [**CharField**](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.CharField) (and its subclasses) require a [**max\_length**](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.CharField.max_length) argument which specifies the size of the **VARCHAR** database field used to store the data.

There’s also a set of common arguments available to all field types. All are optional

**Registering our Models:**

Django comes with an admin.py file which allows us to register our model - basically add tables to our database.

We register our tables with the following code:

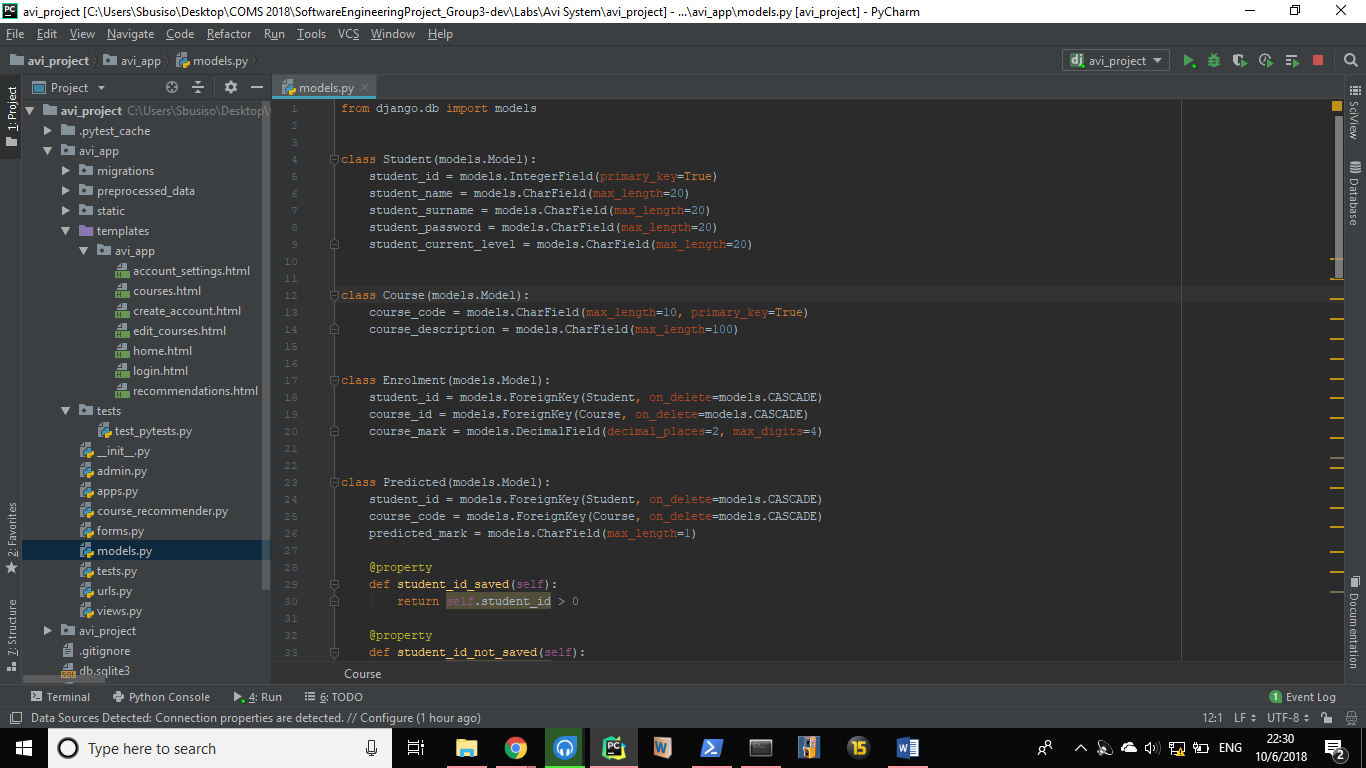


We then use the following commands from our shell to save our database changes:

home\app\_directory:~$ python manage.py makemigrations avi\_app

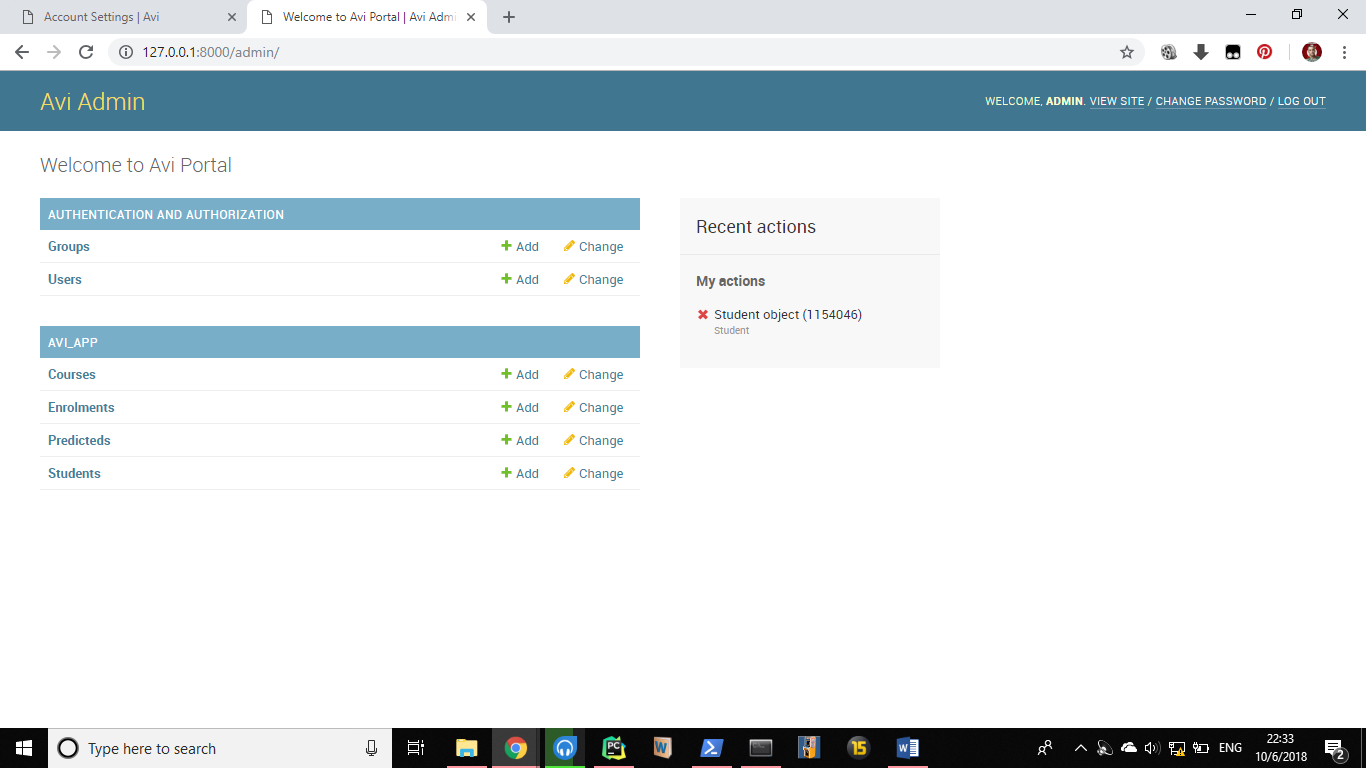
home\app\_directory:~$ python manage.py migrate

Defining and initializing our models(tables):



The shell migration command leads to these tables being created:

(These can be accessed from <http://127.0.0.1:8000/admin> with the admin as username and “pass1234” as password for admin access)



**Django URL Dispatcher:**

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations.

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it’s pure Python code, it can be constructed dynamically.

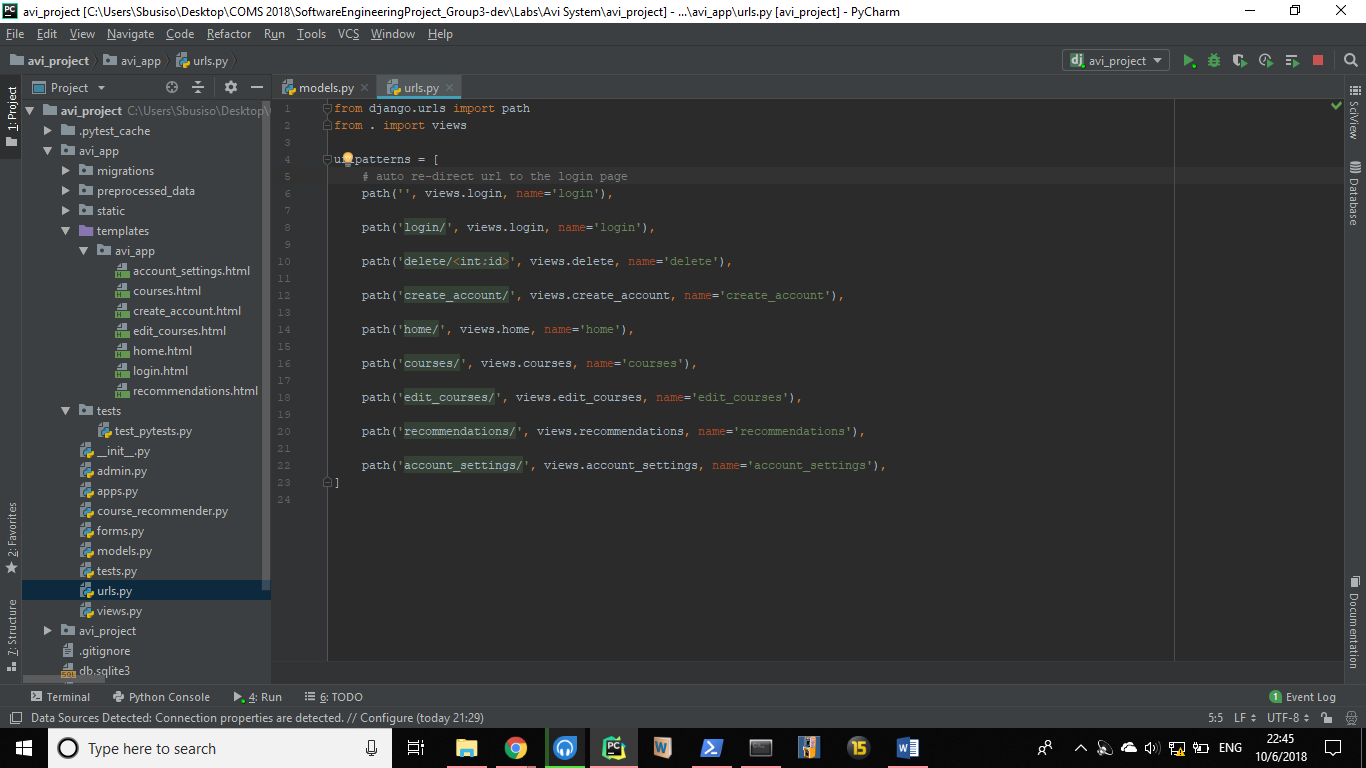
**How Django processes a request**[**¶**](https://docs.djangoproject.com/en/2.1/topics/http/urls/#how-django-processes-a-request)

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django determines the root URLconf module to use. Ordinarily, this is the value of the [**ROOT\_URLCONF**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-ROOT_URLCONF) setting, but if the incoming **HttpRequest** object has a [**urlconf**](https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpRequest.urlconf) attribute (set by middleware), its value will be used in place of the[**ROOT\_URLCONF**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-ROOT_URLCONF) setting.
2. Django loads that Python module and looks for the variable **urlpatterns**. This should be a Python list of [**django.urls.path()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path) and/or [**django.urls.re\_path()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.re_path) instances.
3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.
4. Once one of the URL patterns matches, Django imports and calls the given view, which is a simple Python function (or a [class-based view](https://docs.djangoproject.com/en/2.1/topics/class-based-views/)). The view gets passed the following arguments:
   * An instance of [**HttpRequest**](https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpRequest).
   * If the matched URL pattern returned no named groups, then the matches from the regular expression are provided as positional arguments.
   * The keyword arguments are made up of any named parts matched by the path expression, overridden by any arguments specified in the optional **kwargs** argument to [**django.urls.path()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path) or [**django.urls.re\_path()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.re_path).
5. If no URL pattern matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view.

From “avi\_app/urls.py”:

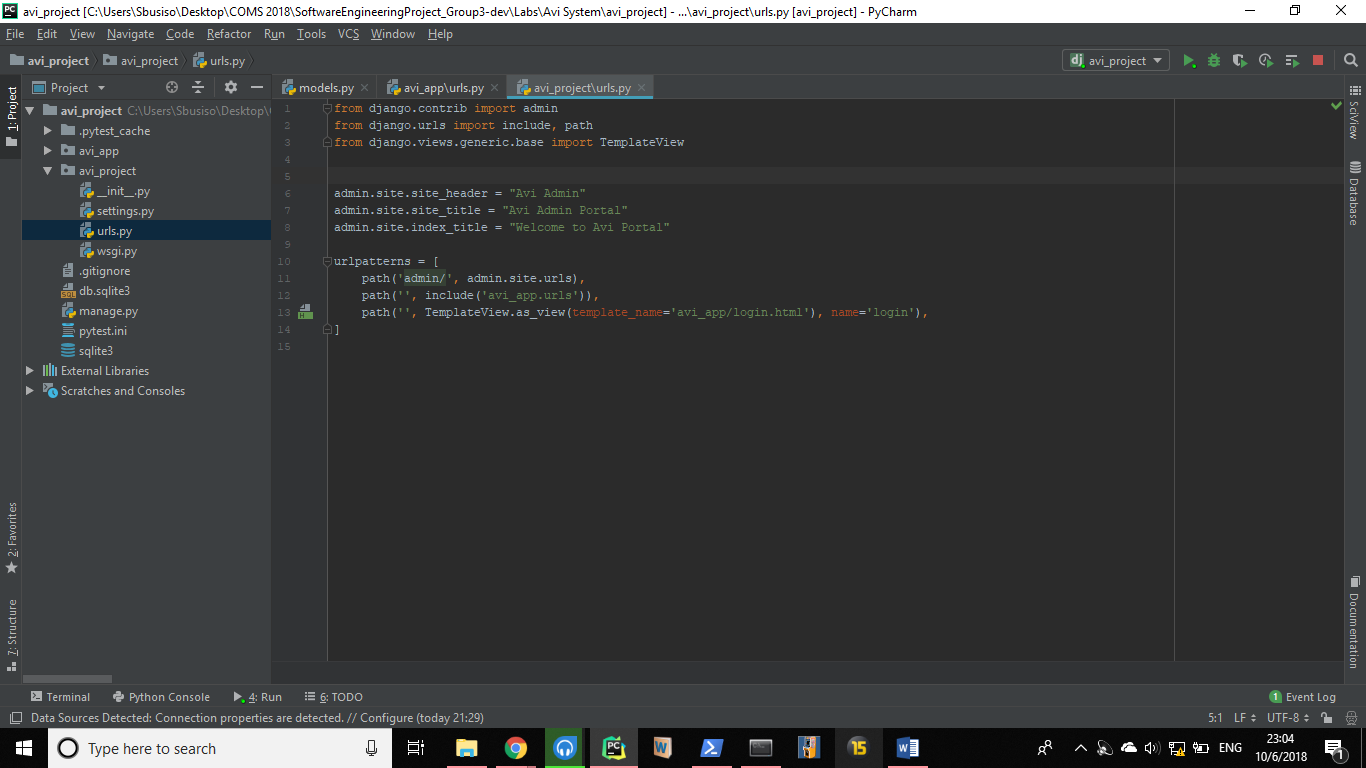
We initialize all urls we want our site to expect, for example localhost:8000/login would be a valid url because we have declared it in line 8.



Line 8 declares the path/url that localhost:8000/login redirects to – views.login then ensures that the login method in the views.py file is associated with this url and used in its execution.

“from . import views “ – allows us to use methods from our views.py file which are then translated to html files and forms which are then displayed to the user.

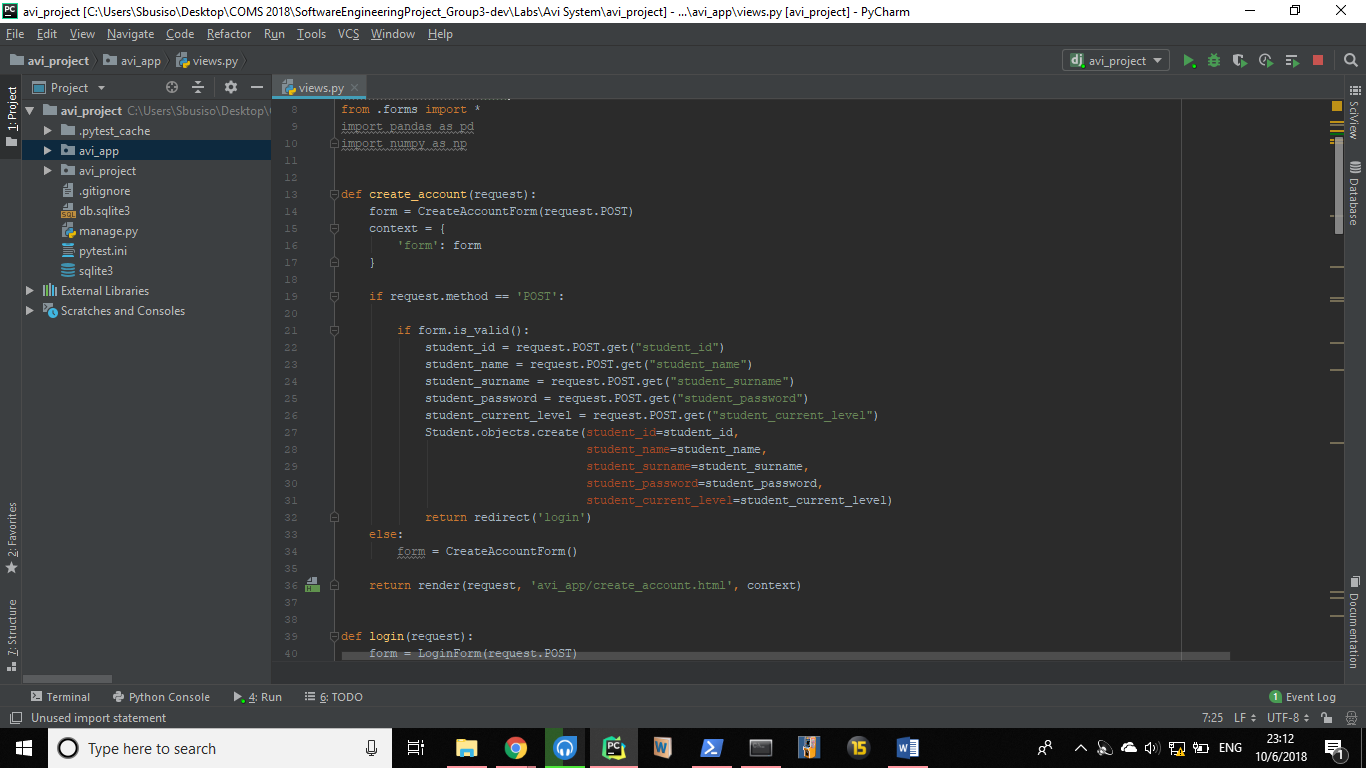
We declare our overall usage of urls in our main url file url.py in line 12 by basically declaring the inclusion of all the other urls in the avi\_app.



**Views:**

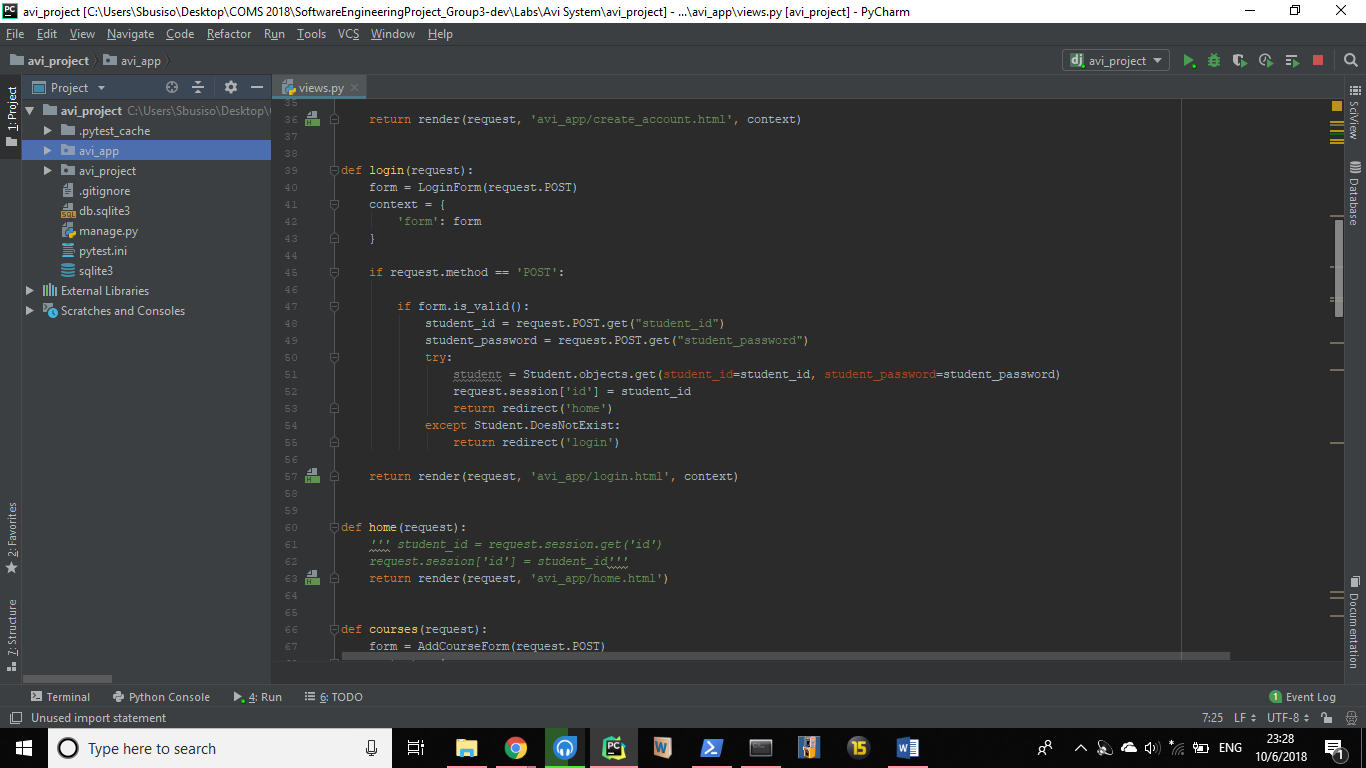
Module by Module break down:

1. Create Account Module:



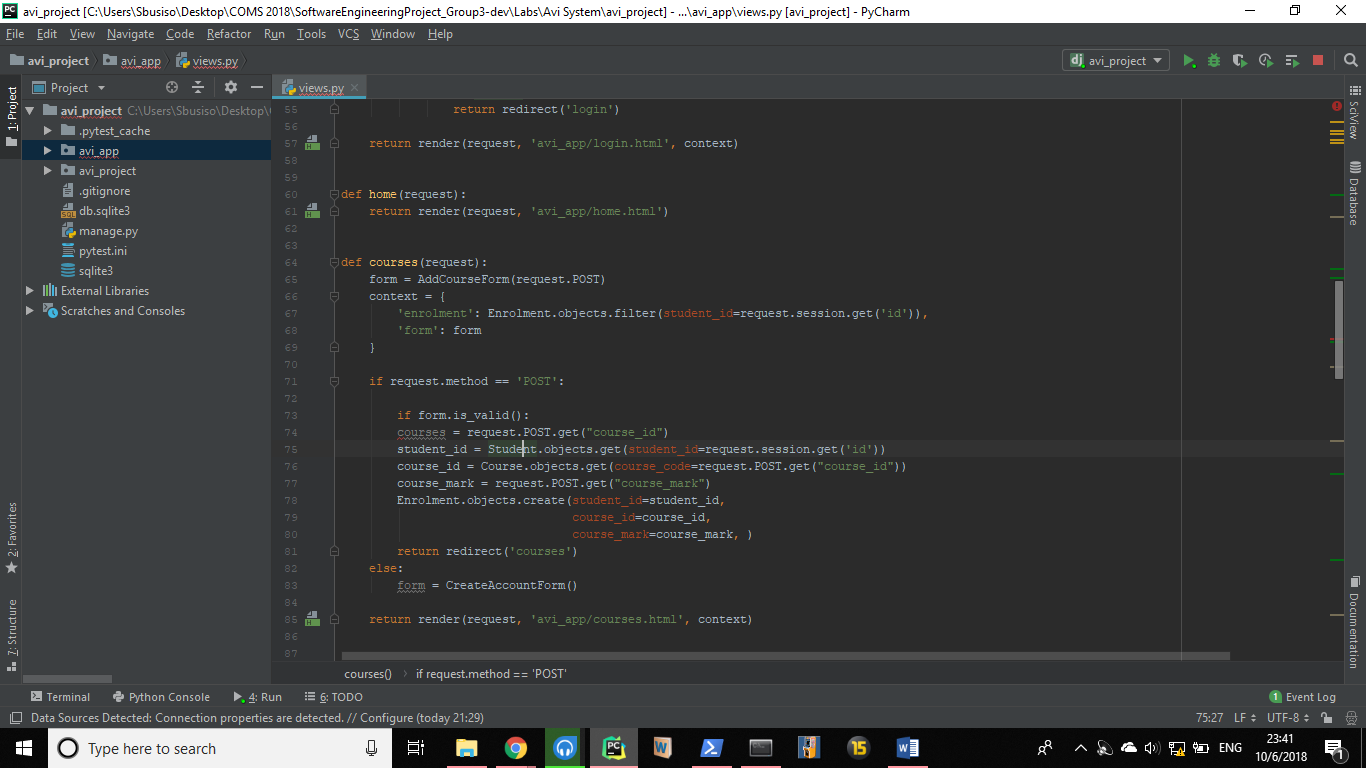
* This module is essential in the create account form functionality.
* It creates a form instance to display (line 13-17).
* Checks if the user is attempting to fill data into the form (line 19).
* If yes, checks if user has submitted valid input to the form (line 21).
* If the form is valid, it then creates and stores the data into variables.
* A student object is then stored in our database table which uses the variables we declared as column attributes.
* It then redirects us to the login page where we are then able to login with our credentials.

1. Login Module



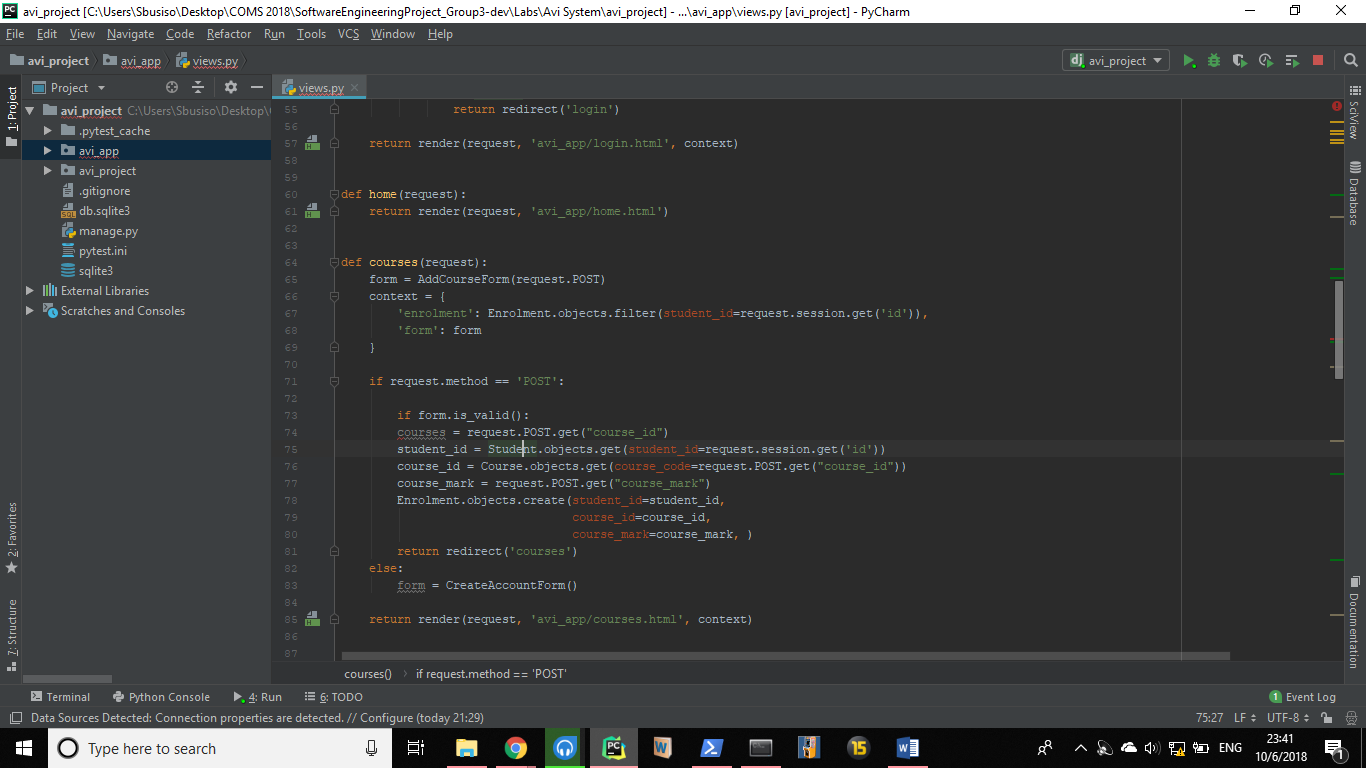
* This module is essential in the create account form functionality.
* It creates a form instance to display (line 40-43).
* Checks if the user is attempting to fill data into the form (line 45).
* If yes, checks if user has submitted valid input to the form (line 47).
* If the form is valid, it then creates variables which are used for authentication by checking if they match the values of the corresponding attribute in our database. (line 50-52)
* If they do, the student is then logged-in and redirected to the home screen. (line 53)
* Else they have to try logging in again. (line 54-55)

1. Home Module:



* This module redirects the request to the homepage when used in urls

1. Course Module:



* This module is essential in the adding courses form functionality.
* It creates a form instance to display (line 65-69).
* Checks if the user is attempting to fill data into the form (line 71).
* If yes, checks if user has submitted valid input to the form (line 73).
* If the form is valid, it then creates and stores the data into variables.
* A course object is then stored in our database table which uses the variables we declared as column attributes.
* It then redirects us to the courses page where we are then able to see our added courses.