MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

Subject Name	**: PYTHON PROGRAMMING**
CLASS	**: I-MCA**
SUBJECT CODE	**: 23PCA13**

Unit V

Django: Installing Django Building an Application Project Creation Designing the Data Schema - Creating an administration site for models - Working with QuerySets and Managers Retrieving Objects Building List and Detail Views.

Django

Django Tutorial provides basic and advanced concepts of Django. Our Django Tutorial is designed for beginners and professionals both.

Django is a Web Application Framework which is used to develop web applications.

Our Django Tutorial includes all topics of Django such as introduction, features, installation, environment setup, admin interface, cookie, form validation, Model, Template Engine, Migration, MVT etc. All the topics are explained in detail so that reader can get enought knowledge of Django.

What is Django

Django is a web application framework written in Python programming language. It is based on MVT (Model View Template) design pattern. The Django is very demanding due to its rapid development feature. It takes less time to build application after collecting client requirement.

This framework uses a famous tag line:**The web framework for perfectionists with deadlines.**

By using Django, we can build web applications in very less time. Django is designed in such a manner that it handles much of configure things automatically, so we can focus on application development only.

History

Django was design and developed by Lawrence journal world in 2003 and publicly released under BSD license in July 2005. Currently, DSF (Django Software Foundation) maintains its development and release cycle.

Django was released on 21, July 2005. Its current stable version is 2.0.3 which was released on 6 March, 2018.

Django Version History

Version	**Date**	**Description**
0.90	16 Nov 2005	
0.91	11 Jan 2006	magic removal
0.96	23 Mar 2007	newforms, testing tools
1.0	3 Sep 2008	API stability, decoupled admin, unicode
1.1	29 Jul 2009	Aggregates, transaction based tests
1.2	17 May 2010	Multiple db connections, CSRF, model validation
1.3	23 Mar 2011	Timezones, in browser testing, app templates.

1.5	26 Feb 2013	Python 3 Support, configurable user model
1.6	6 Nov 2013	Dedicated to Malcolm Tredinnick, db transaction management, connection pooling.
1.7	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	1 Apr 2015	Native support for multiple template engines.*Supported until at least April 2018*
1.9	1 Dec 2015	Automatic password validation. New styling for admin interface.
1.10	1 Aug 2016	Full text search for PostgreSQL. New-style middleware.
1.11 LTS	1.11 LTS	Last version to support Python 2.7.*Supported until at least April 2020*
2.0	Dec 2017	First Python 3-only release, Simplified URL routing syntax, Mobile friendly admin.

Popularity

Django is widely accepted and used by various well-known sites such as:

- Instagram
- Mozilla
- Disqus

- Pinterest
- Bitbucket
- The Washington Times

Features of Django

- Rapid Development
- Secure
- Scalable
- Fully loaded
- Versatile
- Open Source
- Vast and Supported Community

Rapid Development

Django was designed with the intention to make a framework which takes less time to build web application. The project implementation phase is a very time taken but Django creates it rapidly.

Secure

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

Scalable

Django is scalable in nature and has ability to quickly and flexibly switch from small to large scale application project.

Fully loaded

Django includes various helping task modules and libraries which can be used to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds etc.

Versatile

Django is versatile in nature which allows it to build applications for different-different domains. Now a days, Companies are using Django to build various types of applications like: content management systems, social networks sites or scientific computing platforms etc.

Open Source

Django is an open source web application framework. It is publicly available without cost. It can be downloaded with source code from the public repository. Open source reduces the total cost of the application development.

Vast and Supported Community

Django is an one of the most popular web framework. It has widely supportive community and channels to share and connect.

Django Installation

To install Django, first visit to **django official site (https://www.djangoproject.com)** and download django by clicking on the download section. Here, we will see various options to download The Django.

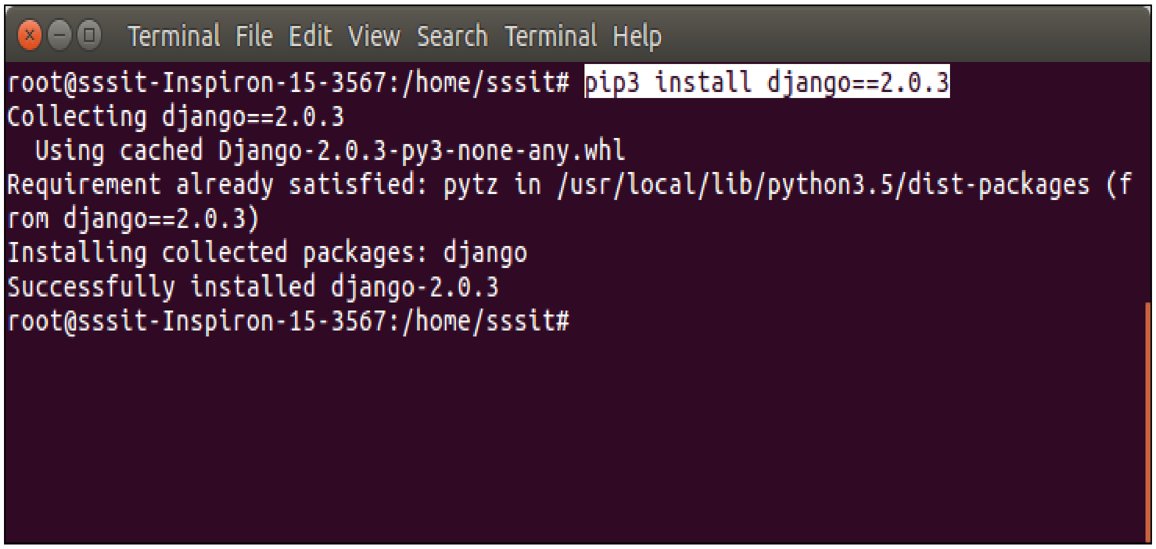
Django requires **pip** to start installation. Pip is a package manager system which is used to install and manage packages written in python. For Python 3.4 and higher versions **pip3** is used to manage packages.

we are installing Django in Ubuntu operating system.

The complete installation process is described below. Before installing make sure **pip is installed** in local system.

Here, we are installing Django using pip3, the installation command is given below.

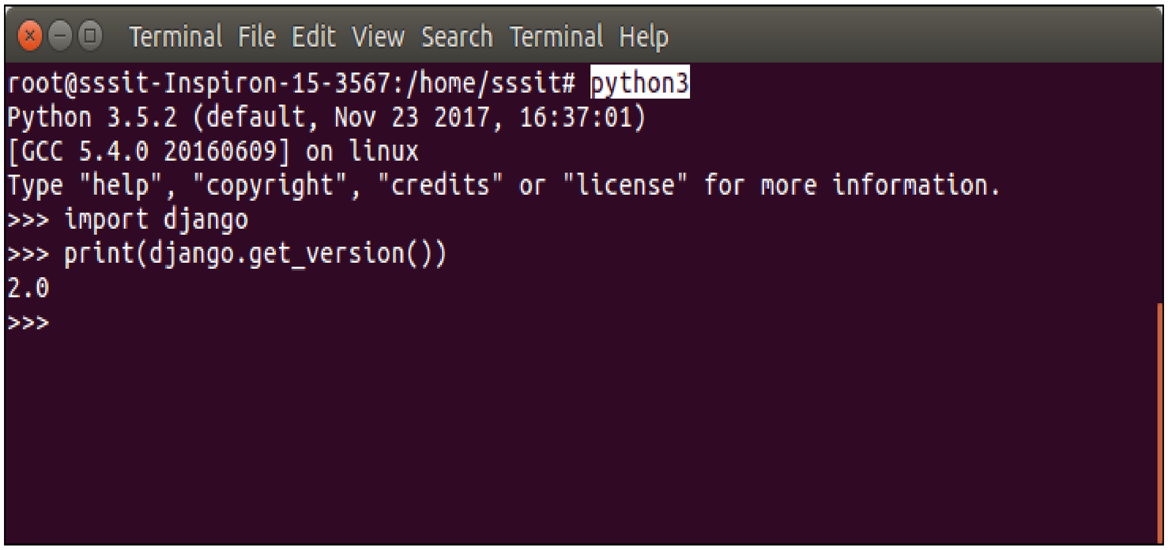
1. $ pip3 install django==2.0.3



```
root@sssit-Inspiron-15-3567:/home/sssit# pip3 install django==2.0.3
Collecting django==2.0.3
  Using cached Django-2.0.3-py3-none-any.whl
Requirement already satisfied: pytz in /usr/local/lib/python3.5/dist-packages (f
rom django==2.0.3)
Installing collected packages: django
Successfully installed django-2.0.3
root@sssit-Inspiron-15-3567:/home/sssit#
```

Verify Django Installation

After installing Django, we need to varify the installation. Open terminal and write **python3** and press enter. It will display python shell where we can verify django installation.



```
root@sssit-Inspiron-15-3567:/home/sssit# python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> print(django.get_version())
2.0
>>>
```

Look at the Django version displayed by the print method of the python. Well, Django is installed successfuly. Now, we can build Django web applications.

create a Django project

Dive into the world of web development with Python by exploring the versatile Django framework. Django is a go-to for many developers due to its popularity, open-source license, and robust security features. It enables fast and efficient project development. In this tutorial, we will guide you through the process of installing Django on a Windows machine using pip, verifying the installation, creating a new project, and launching a Django development server. Get ready to enhance your web development skills and experience the power of Django.

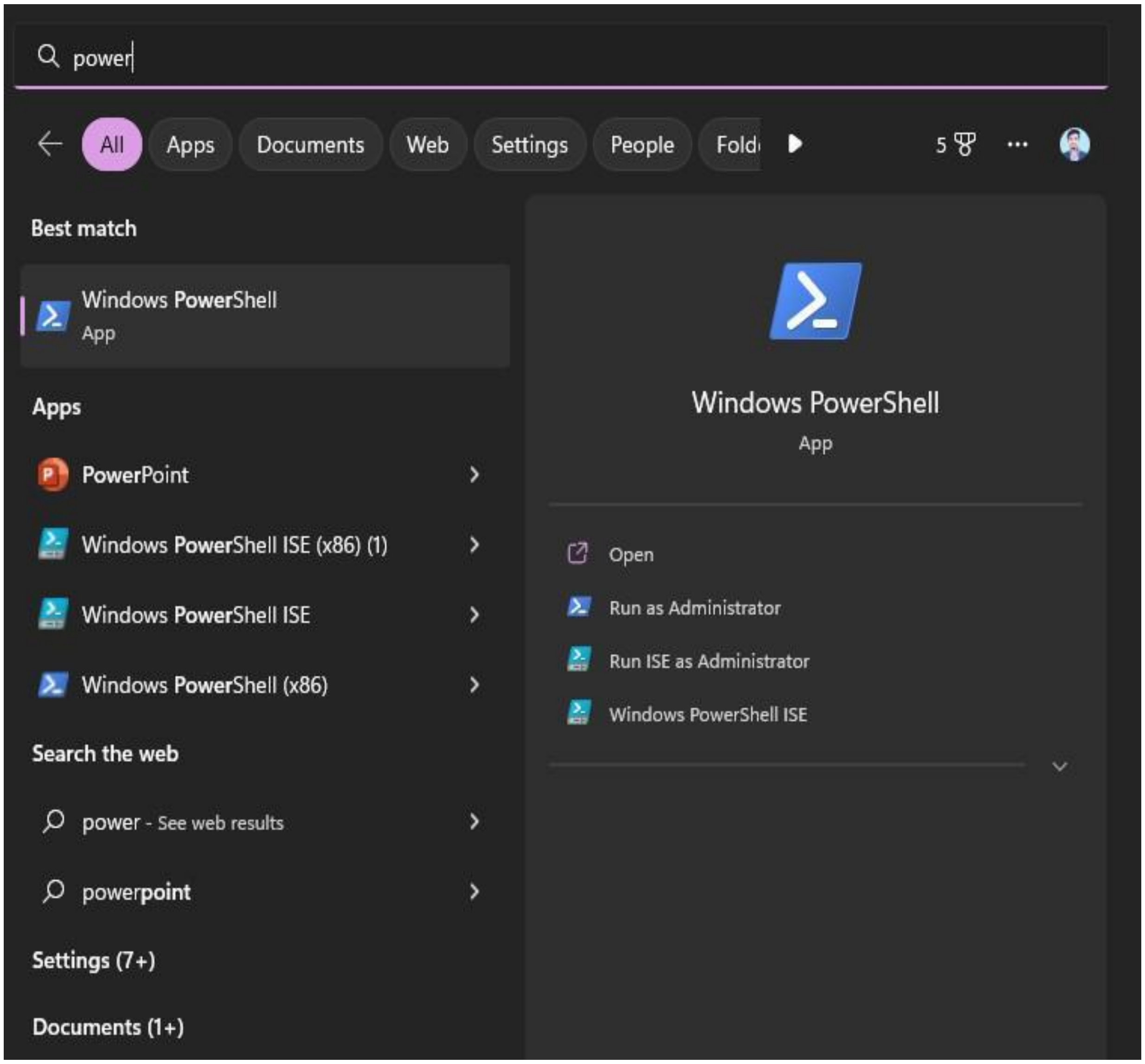
Create a Project in Django

Before you begin, you will need to have the following:

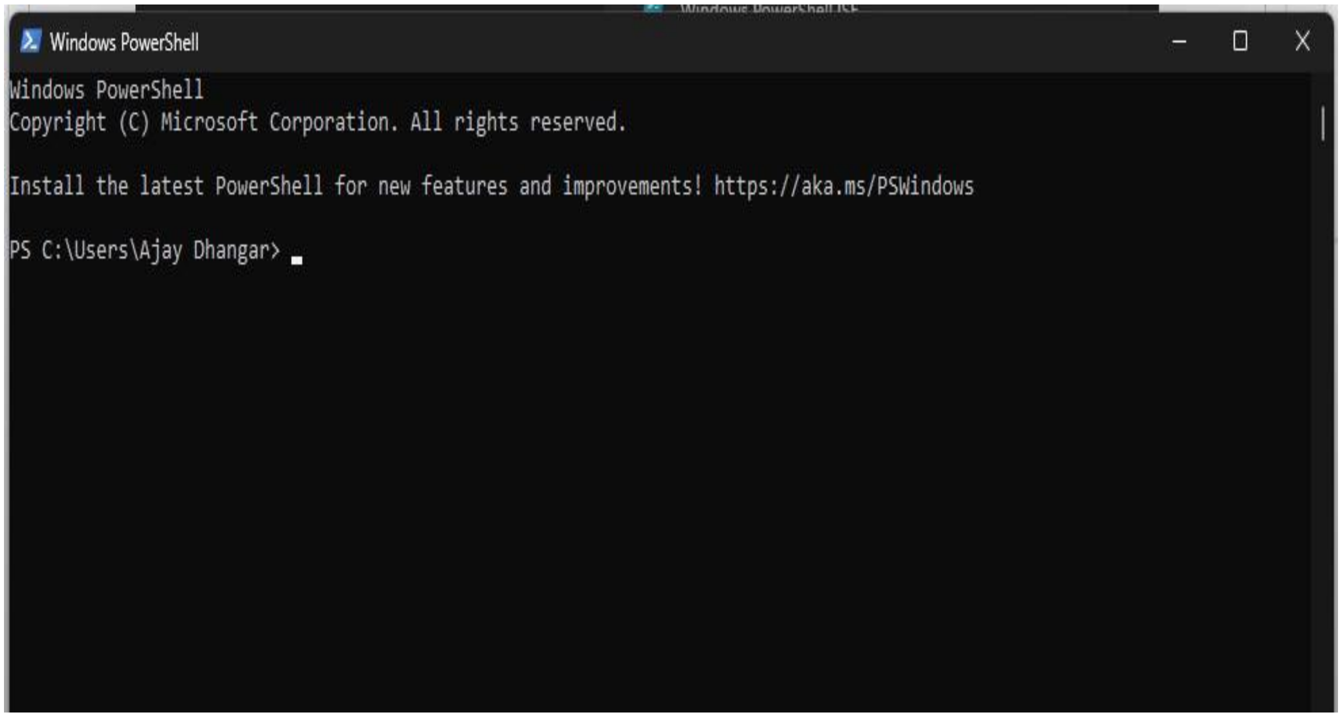
- Windows 10 or later
- Python 3.6 or later
- Python pip

Step 1: Opening PowerShell

The first step is to open PowerShell. To do this, press the Windows key and type **PowerShell**. Right-click on the PowerShell icon and select **Run as Administrator.**


power
All
Apps
Documents
Web
Settings
People
Fold
5
Best match
Windows PowerShell
App
Apps
PowerPoint
Windows PowerShell ISE (x86) (1)
Windows PowerShell ISE
Windows PowerShell (x86)
Search the web
power - See web results
powerpoint
Settings (7+)
Documents (1+)
Windows PowerShell
App
Open
Run as Administrator
Run ISE as Administrator
Windows PowerShell ISE

PowerShell



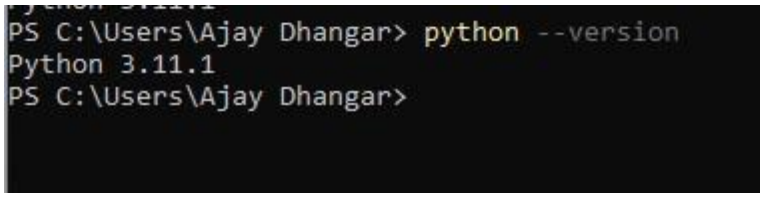
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Ajay Dhangar>
```

Step 2: Verifying Python Installation

Once PowerShell is open, you can verify that Python is installed by typing ***python –version***. This will display the version of Python that is installed.

Terminal command:

python --version

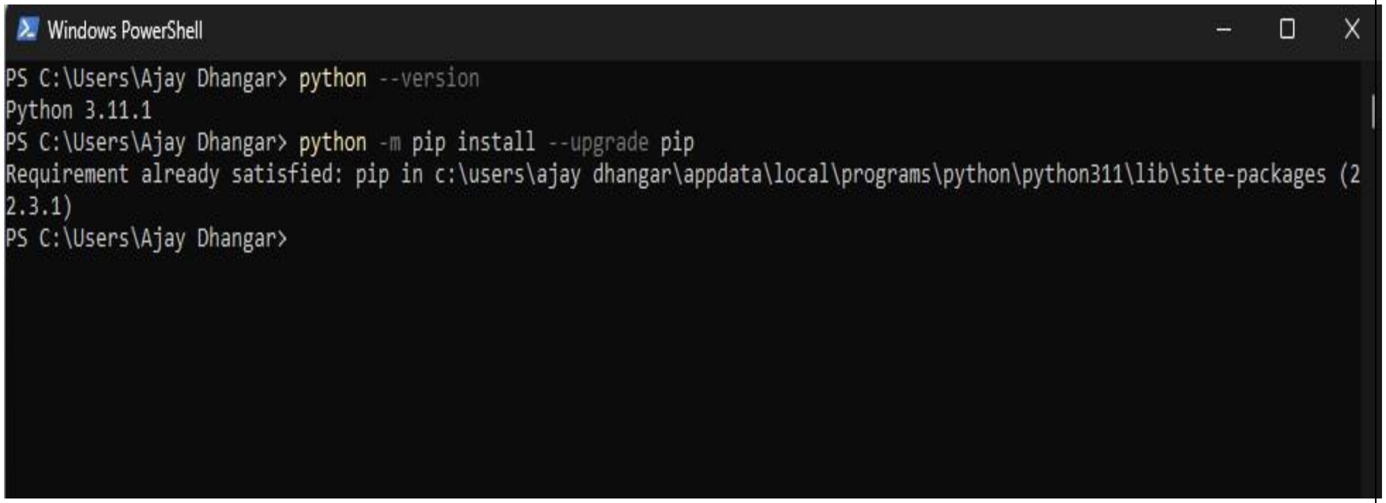


```
PS C:\Users\Ajay Dhangar> python --version
Python 3.11.1
PS C:\Users\Ajay Dhangar>
```

Python is installed

Step 3: Upgrading PIP

Next, you will need to upgrade *pip*. To do this, type ***python -m pip install –upgrade pip***. This will ensure that you have the latest version of pip installed.



```
PS C:\Users\Ajay Dhangar> python --version
Python 3.11.1
PS C:\Users\Ajay Dhangar> python -m pip install --upgrade pip
Requirement already satisfied: pip in c:\users\ajay dhangar\appdata\local\programs\python\python311\lib\site-packages (2
2.3.1)
PS C:\Users\Ajay Dhangar>
```

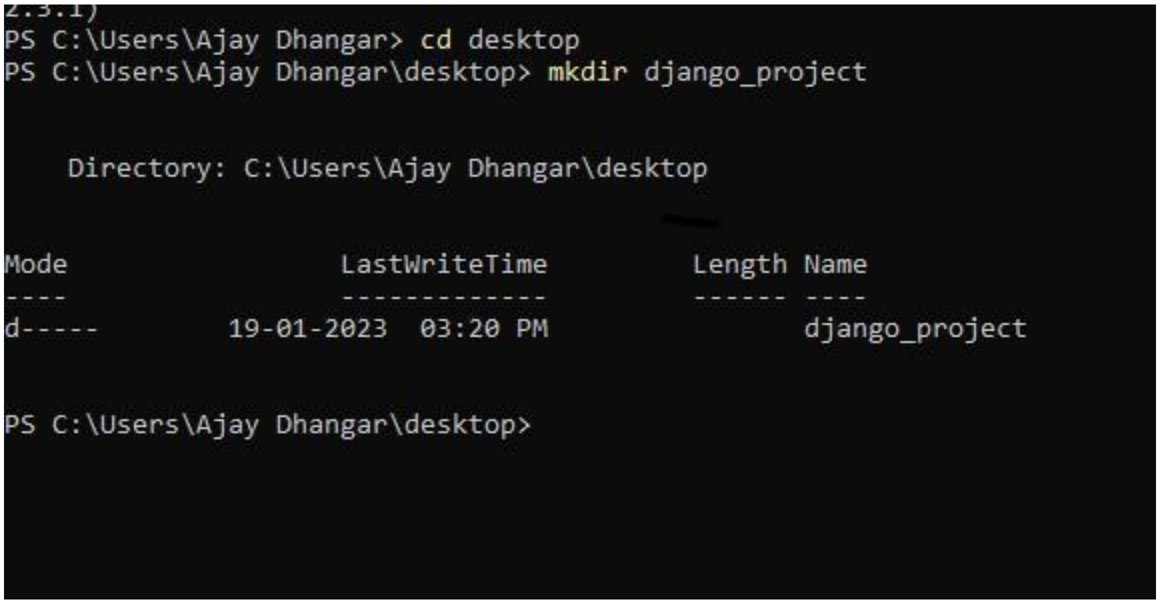
upgrade pip

Step 4: Creating a Project Directory

Now, you will need to create a project directory. To do this, type **mkdir myproject**. This will create a directory called **myproject.**

(i) Create the directory by utilizing the **mkdir** command. This command is a powerful tool that allows you to quickly and easily create a directory in your system. With this command, you can create a directory in seconds, making it a great time-saver for any user.

mkdir django_project

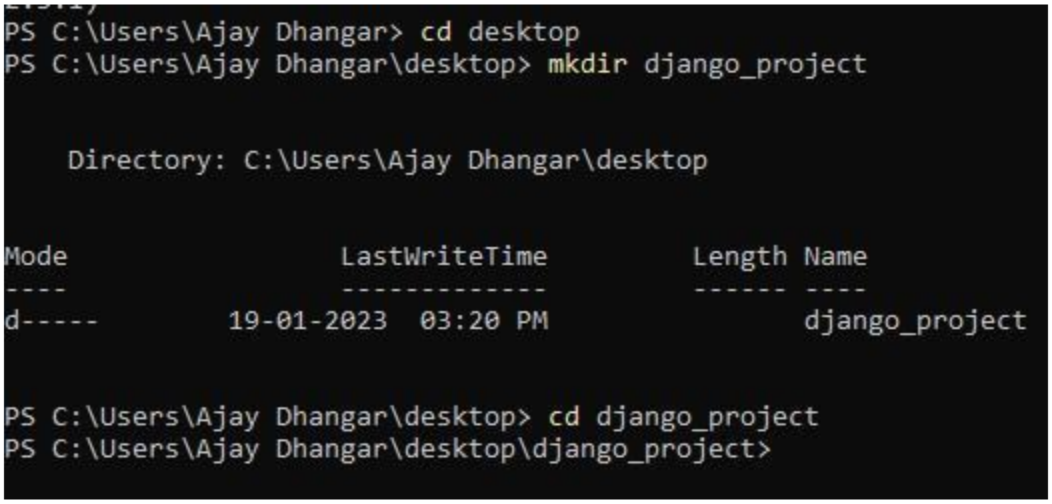


```
2.3.1)
PS C:\Users\Ajay Dhangar> cd desktop
PS C:\Users\Ajay Dhangar\desktop> mkdir django_project


    Directory: C:\Users\Ajay Dhangar\desktop


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        19-01-2023  03:20 PM                django_project


PS C:\Users\Ajay Dhangar\desktop>
```

Create the directory

(ii) Navigate to the ***Django_Project*** directory using the `` `cd` `` command. Here, you'll find all the necessary components to get your project up and running.

cd django_project



```
PS C:\Users\Ajay Dhangar> cd desktop
PS C:\Users\Ajay Dhangar\desktop> mkdir django_project


    Directory: C:\Users\Ajay Dhangar\desktop


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        19-01-2023  03:20 PM                django_project


PS C:\Users\Ajay Dhangar\desktop> cd django_project
PS C:\Users\Ajay Dhangar\desktop\django_project>
```

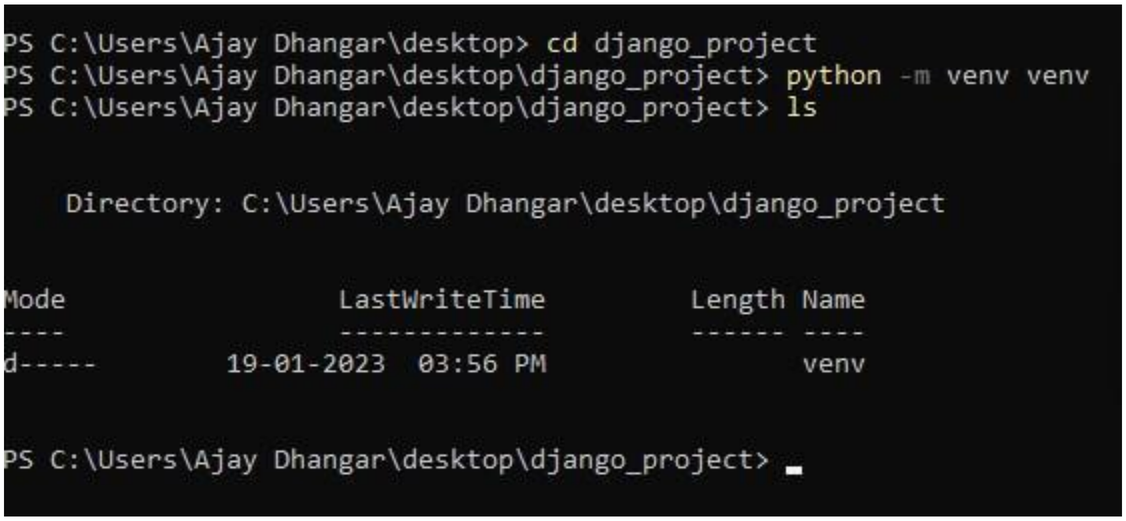
Django_Project directory

Step 5: Creating the Virtual Environment

Next, you will need to create a virtual environment. To create a virtual environment called **myproject**.

To do this, type in terminal

python -m venv myproject



```
PS C:\Users\Ajay Dhangar\desktop> cd django_project
PS C:\Users\Ajay Dhangar\desktop\django_project> python -m venv venv
PS C:\Users\Ajay Dhangar\desktop\django_project> ls


    Directory: C:\Users\Ajay Dhangar\desktop\django_project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        19-01-2023  03:56 PM                venv


PS C:\Users\Ajay Dhangar\desktop\django_project>
```

python -m venv myproject and ls

Step 6: Activating the Virtual Environment

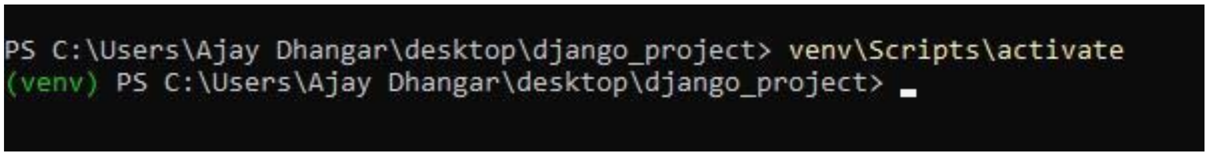
Now, you will need to activate the virtual environment. To do this, type

Step 6: Activating the Virtual Environment

Now, you will need to activate the virtual environment. To do this, type

0 seconds of 15 secondsVolume 0%

venv\Scripts\activate



```
PS C:\Users\Ajay Dhangar\desktop\django_project> venv\Scripts\activate
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project>
```

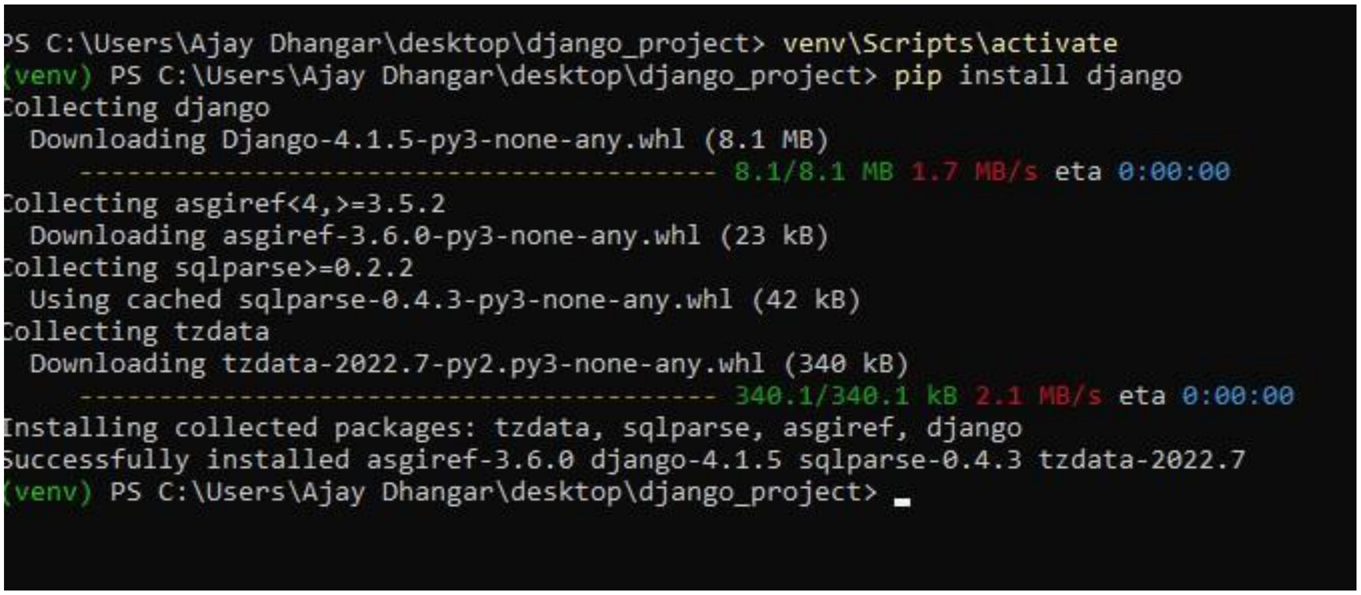
activate the virtual environment

Step 7: Installing Django

This section will teach you how to install Django on your system using ***pip***. To do this, run the following command:

pip install Django

This will install the latest version of Django, allowing you to start building powerful web applications easily.

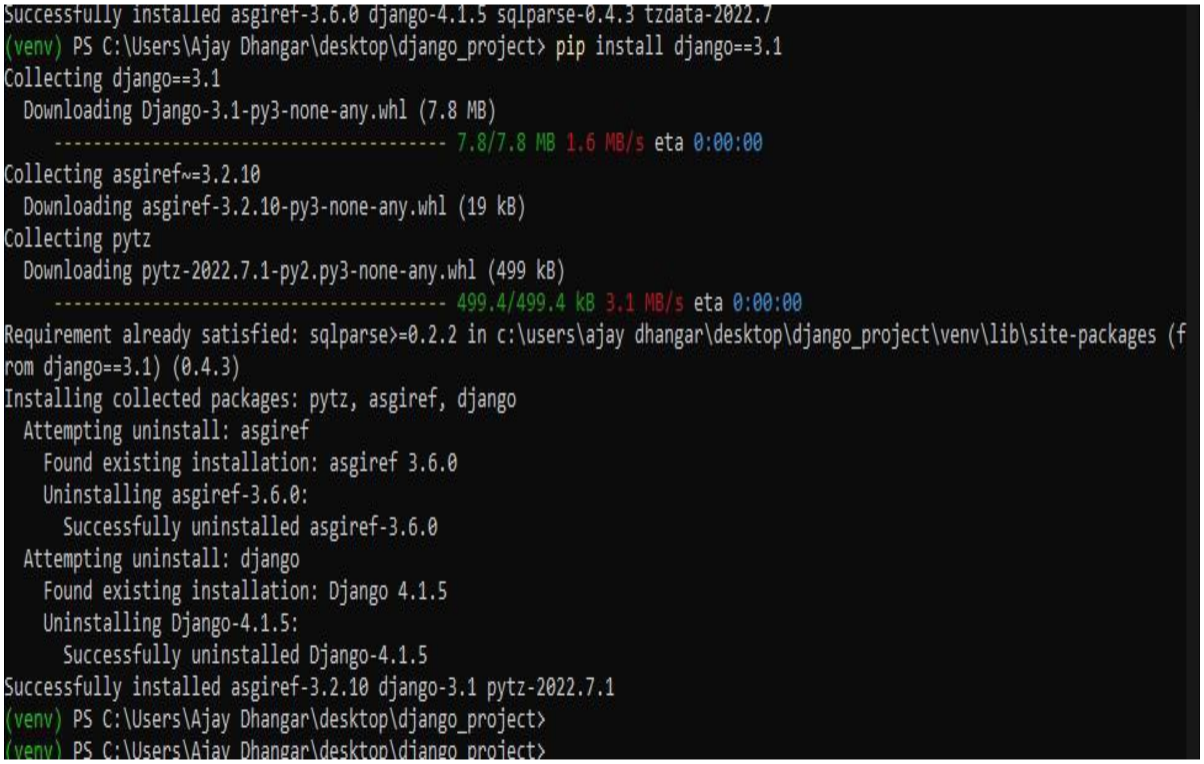


```
PS C:\Users\Ajay Dhangar\desktop\django_project> venv\Scripts\activate
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project> pip install django
Collecting django
  Downloading Django-4.1.5-py3-none-any.whl (8.1 MB)
     ---------------------------------------- 8.1/8.1 MB 1.7 MB/s eta 0:00:00
Collecting asgiref<4,>=3.5.2
  Downloading asgiref-3.6.0-py3-none-any.whl (23 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.3-py3-none-any.whl (42 kB)
Collecting tzdata
  Downloading tzdata-2022.7-py2.py3-none-any.whl (340 kB)
     ---------------------------------------- 340.1/340.1 kB 2.1 MB/s eta 0:00:00
Installing collected packages: tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.6.0 django-4.1.5 sqlparse-0.4.3 tzdata-2022.7
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project>
```

Installing Django

If you wish to install a different version of Django, you can easily specify the version you desire by following these steps:

pip install django

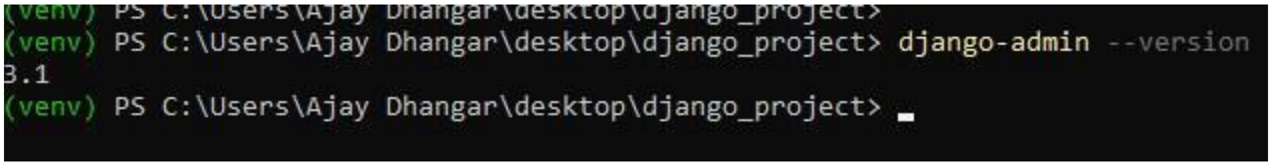


```
Successfully installed asgiref-3.6.0 django-4.1.5 sqlparse-0.4.3 tzdata-2022.7
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project> pip install django==3.1
Collecting django==3.1
  Downloading Django-3.1-py3-none-any.whl (7.8 MB)
     ---------------------------------------- 7.8/7.8 MB 1.6 MB/s eta 0:00:00
Collecting asgiref~=3.2.10
  Downloading asgiref-3.2.10-py3-none-any.whl (19 kB)
Collecting pytz
  Downloading pytz-2022.7.1-py2.py3-none-any.whl (499 kB)
     ---------------------------------------- 499.4/499.4 kB 3.1 MB/s eta 0:00:00
Requirement already satisfied: sqlparse>=0.2.2 in c:\users\ajay dhangar\desktop\django_project\venv\lib\site-packages (f
rom django==3.1) (0.4.3)
Installing collected packages: pytz, asgiref, django
  Attempting uninstall: asgiref
    Found existing installation: asgiref 3.6.0
    Uninstalling asgiref-3.6.0:
      Successfully uninstalled asgiref-3.6.0
  Attempting uninstall: django
    Found existing installation: Django 4.1.5
    Uninstalling Django-4.1.5:
      Successfully uninstalled Django-4.1.5
Successfully installed asgiref-3.2.10 django-3.1 pytz-2022.7.1
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project>
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project>
```

version of Django

Once the installation is complete, you must verify that Django has been successfully installed. To do this, enter the following command:

django-admin --version



```
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project>
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project> django-admin --version
3.1
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project>
```

Django version

Step 8: Create the Django Project

Now it’s time to create a project. According to the Django documentation, a project is a Python package – i.e. a directory of code – that contains all the settings for an instance of Django. This includes database configuration, Django-specific options, and application-specific settings.

To create the project, use the command-line utility *django-admin* that comes with Django. This command will generate files where you can configure the settings for your database, add third-party packages, and more.

Create the project using the following command:

django-admin startproject test_project

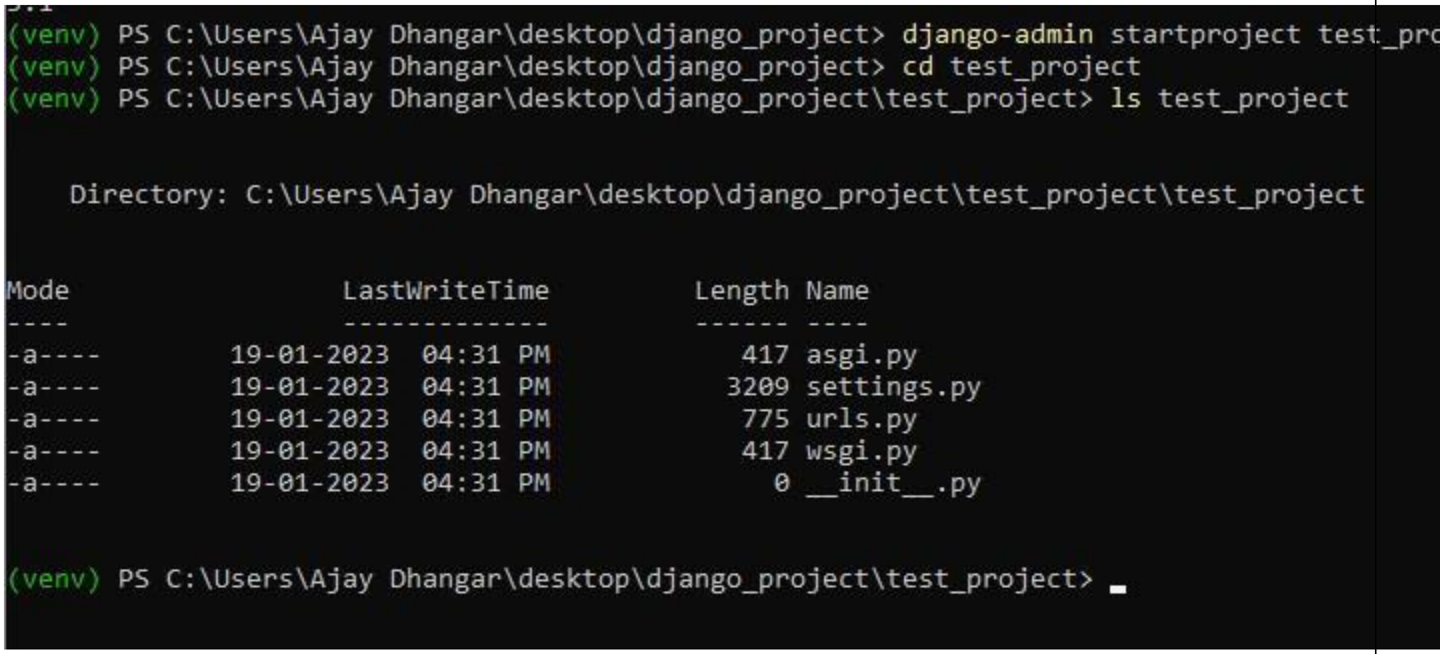
Change into the ***test_project*** directory:

cd test_project

Type the following command to see the contents in the project directory:

ls test_project

Output:



```
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project> django-admin startproject test_pro
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project> cd test_project
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project\test_project> ls test_project


    Directory: C:\Users\Ajay Dhangar\desktop\django_project\test_project\test_project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        19-01-2023  04:31 PM            417 asgi.py
-a----        19-01-2023  04:31 PM           3209 settings.py
-a----        19-01-2023  04:31 PM            775 urls.py
-a----        19-01-2023  04:31 PM            417 wsgi.py
-a----        19-01-2023  04:31 PM              0 __init__.py


(venv) PS C:\Users\Ajay Dhangar\desktop\django_project\test_project>
```

output

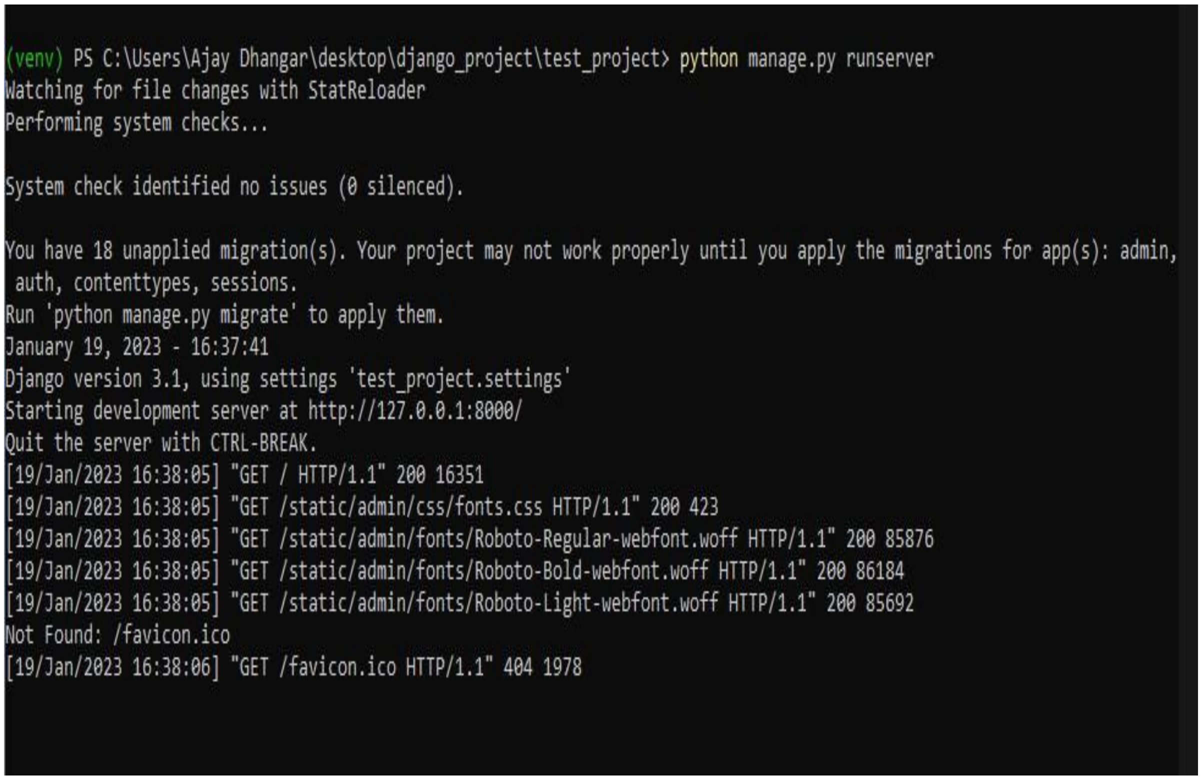
The directory ***test_project*** contains Django configuration files. The manage.py file is especially useful when starting a development server, which is what you will do in the next step.

Step 9: Running the Development Server

Now that the project has been created, it’s time to start the Django development server.Start the development server using the ***manage.py runserver*** command:

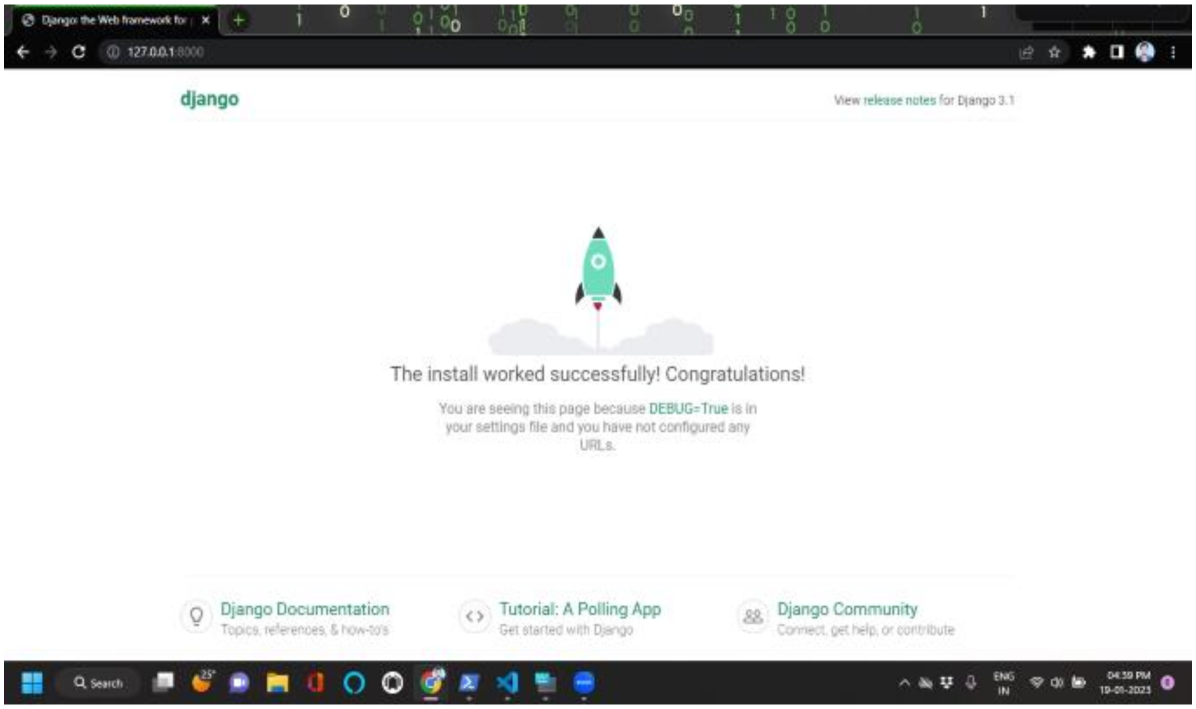
Note: Make sure sure you are in the same directory as manage.py file after to run manage.py runserver

python manage.py runserver



```
(venv) PS C:\Users\Ajay Dhangar\desktop\django_project\test_project> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
January 19, 2023 - 16:37:41
Django version 3.1, using settings 'test_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[19/Jan/2023 16:38:05] "GET / HTTP/1.1" 200 16351
[19/Jan/2023 16:38:05] "GET /static/admin/css/fonts.css HTTP/1.1" 200 423
[19/Jan/2023 16:38:05] "GET /static/admin/fonts/Roboto-Regular-webfont.woff HTTP/1.1" 200 85876
[19/Jan/2023 16:38:05] "GET /static/admin/fonts/Roboto-Bold-webfont.woff HTTP/1.1" 200 86184
[19/Jan/2023 16:38:05] "GET /static/admin/fonts/Roboto-Light-webfont.woff HTTP/1.1" 200 85692
Not Found: /favicon.ico
[19/Jan/2023 16:38:06] "GET /favicon.ico HTTP/1.1" 404 1978
```

Start



Django: the Web framework for
127.0.0.1:8000
django
View release notes for Django 3.1

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute
04:39 PM
19-01-2023

Step 10: Create app in django

Now as we created the project in django and our django server is up and running and if now to create app in django project we can achieve it b by using django inbuit command which will initiate the django app for us.

Note: Make sure before running the command we must inside our project directory if not then run the below command

cd test_project

now we are inside the test_project and we are ready to create django app

django-admin startapp test_app

Now this command will create app inside our project and we ready to go with the django project and to know more about django app you can refer to the article – How to create app in django.

Django QuerySet

Designing the blog data schema

You will start designing your blog data schema by defining the data models for your blog. A model is a Python class that subclasses django.db.models.Model in which each attribute represents a database field. Django will create a table for each model defined in the models.py file. When you create a model, Django will provide you with a practical API to query objects in the database easily.

First, you need to define a Post model. Add the following lines to the models.py file of the blog application:

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    author = models.ForeignKey(User,
                              on_delete=models.CASCADE,
                              related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
                              choices=STATUS_CHOICES,
                              default='draft')
    class Meta:
        ordering = ('-publish',)
    def __str__(self):
        return self.title
```
Copy

This is your data model for blog posts. Let's take a look at the fields you just defined for this model:

- title: This is the field for the post title. This field is CharField, which translates into a VARCHAR column in the SQL database.
- slug: This is a field intended to be used in URLs. A slug is a short label that contains only letters, numbers, underscores, or hyphens. You will use the slug field to build beautiful, SEO-friendly URLs for your blog posts. You have added the unique_for_date parameter to this field so that you can build URLs for posts using their publish date and slug. Django will prevent multiple posts from having the same slug for a given date.

- author: This field defines a many-to-one relationship, meaning that each post is written by a user, and a user can write any number of posts. For this field, Django will create a foreign key in the database using the primary key of the related model. In this case, you are relying on the User model of the Django authentication system.
  The on_delete parameter specifies the behavior to adopt when the referenced object is deleted. This is not specific to Django; it is an SQL standard. Using CASCADE, you specify that when the referenced user is deleted, the database will also delete all related blog posts. You can take a look at all the possible options at https://docs.djangoproject.com/en/3.0/ref/models/fields/#django.db.models.ForeignKey.on_delete. You specify the name of the reverse relationship, from User to Post, with the related_name attribute. This will allow you to access related objects easily. You will learn more about this later.
- body: This is the body of the post. This field is a text field that translates into a TEXT column in the SQL database.
- publish: This datetime indicates when the post was published. You use Django's timezone now method as the default value. This returns the current datetime in a timezone-aware format. You can think of it as a timezone-aware version of the standard Python datetime.now method.
- created: This datetime indicates when the post was created. Since you are using auto_now_add here, the date will be saved automatically when creating an object.
- updated: This datetime indicates the last time the post was updated. Since you are using auto_now here, the date will be updated automatically when saving an object.
- status: This field shows the status of a post. You use a choices parameter, so the value of this field can only be set to one of the given choices.

Django comes with different types of fields that you can use to define your models. You can find all field types at https://docs.djangoproject.com/en/3.0/ref/models/fields/.

The Meta class inside the model contains metadata. You tell Django to sort results by the publish field in descending order by default when you query the database. You specify the descending order using the negative prefix. By doing this, posts published recently will appear first.

The __str__() method is the default human-readable representation of the object. Django will use it in many places, such as the administration site.

If you are coming from using Python 2.x, note that in Python 3, all strings are natively considered Unicode; therefore, we only use the __str__() method and the __unicode__() method is obsolete.

Activating the application

In order for Django to keep track of your application and be able to create database tables for its models, you have to activate it. To do this, edit the settings.py file and add blog.apps.BlogConfig to the INSTALLED_APPS setting. It should look like this:

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

Copy

The BlogConfig class is your application configuration. Now Django knows that your application is active for this project and will be able to load its models.

Creating and applying migrations

Now that you have a data model for your blog posts, you will need a database table for it. Django comes with a migration system that tracks the changes made to models and enables them to propagate into the database. As mentioned, the migrate command applies migrations for all applications listed in INSTALLED_APPS; it synchronizes the database with the current models and existig migrations.

First, you will need to create an initial migration for your Post model. In the root directory of your project, run the following command:

```
python manage.py makemigrations blog
```

Copy

You should get the following output:

```
Migrations for 'blog':

  blog/migrations/0001_initial.py

    - Create model Post
```

Copy

Django just created the 0001_initial.py file inside the migrations directory of the blog application. You can open that file to see how a migration appears. A migration specifies dependencies on other migrations and operations to perform in the database to synchronize it with model changes.

Let's take a look at the SQL code that Django will execute in the database to create the table for your model. The sqlmigrate command takes the migration names and returns their SQL without executing it. Run the following command to inspect the SQL output of your first migration:

```
python manage.py sqlmigrate blog 0001
```

Copy

The output should look as follows:

```
BEGIN;

--

-- Create model Post

--

CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY

AUTOINCREMENT, "title" varchar(250) NOT NULL, "slug" varchar(250) NOT NULL,

"body" text NOT NULL, "publish" datetime NOT NULL, "created" datetime NOT NULL,

"updated" datetime NOT NULL, "status" varchar(10) NOT NULL, "author_id" integer NOT

NULL REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY DEFERRED);

CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");

CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");

COMMIT;
```

Copy

The exact output depends on the database you are using. The preceding output is generated for SQLite. As you can see in the output, Django generates the table names by combining the application name and the lowercase name of the model (blog_post), but you can also specify a custom database name for your model in the Meta class of the model using the db_table attribute.

Django creates a primary key automatically for each model, but you can also override this by specifying primary_key=True in one of your model fields. The default primary key is an id column, which consists of an integer that is incremented automatically. This column corresponds to the id field that is automatically added to your models.

Let's sync your database with the new model. Run the following command to apply existing migrations:

```
python manage.py migrate
```

Copy

You will get an output that ends with the following line:

```
Applying blog.0001_initial... OK
```

Copy

You just applied migrations for the applications listed in INSTALLED_APPS, including your blog application. After applying the migrations, the database reflects the current status of your models.

If you edit the models.py file in order to add, remove, or change the fields of existing models, or if you add new models, you will have to create a new migration using the makemigrations command. The migration will allow Django to keep track of model changes. Then, you will have to apply it with the migrate command to keep the database in sync with your models.

Django Admin With Python

The Django **framework comes with a powerful administrative tool called** admin**. You can use it out of the box to quickly add, delete, or edit any database model from a web interface. But with a little extra code, you can customize the Django admin to take your admin capabilities to the next level.**

- Add **attribute columns** in the model object list
- **Link** between model objects
- Add **filters** to the model object list
- Make model object lists **searchable**
- Modify the object **edit forms**
- Override Django **admin templates**

Prerequisites

To get the most out of this tutorial, you'll need some familiarity with Django, particularly model objects. As Django isn't part of the standard Python library, it's best if you also have some knowledge of pip and pyenv (or an equivalent virtual environment tool). To learn more about these topics, check out the following resources:

- Get Started With Django Part 1: Build a Portfolio App
- What is Pip? A Guide for New Pythonistas
- Managing Multiple Python Versions With pyenv
- What Virtual Environments Are Good For

You may also be interested in one of the many available Django tutorials.

The code snippets in this tutorial were tested against Django 3.0.7. All the concepts predate Django 2.0, so they should work in whatever version you're using, but minor differences may exist.

Setting Up the Django Admin

The **Django admin** provides a web-based interface for creating and managing database model objects. To see it in action, you’ll first need a Django project and some object models. Install Django inside a clean virtual environment:

```
$ python -m pip install django
$ django-admin startproject School
$ cd School
$ ./manage.py startapp core
$ ./manage.py migrate
$ ./manage.py createsuperuser
Username: admin
Email address: admin@example.com
Password:
Password (again):
```

You first create a new Django project named School with an app called core. Then you migrate the authentication tables and create an administrator. Access to the Django admin screens is restricted to users with staff or superuser flags, so you use the createsuperuser management command to create a superuser.

You also need to modify School/settings.py to include the new app named core:

```
# School/settings.py
# ...

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "core",    # Add this line
]
```

The core app directory will start with the following files inside:

```
core/
│
├── migrations/
│   └── __init__.py
│
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
└── views.py
```

You’re interested in two of these files:

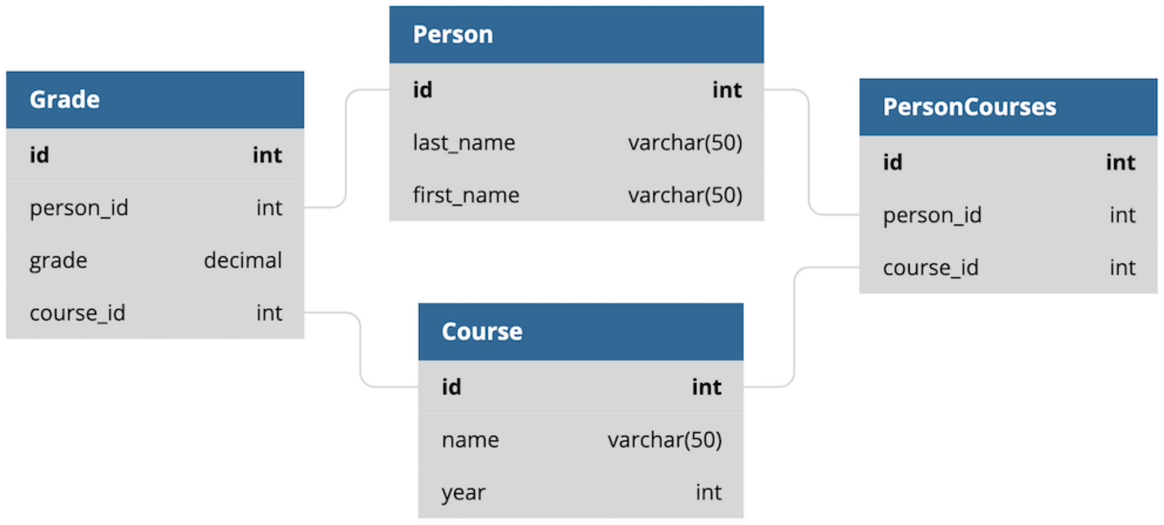
1. **models.py** defines your database models.
2. **admin.py** registers your models with the Django admin.

To demonstrate the outcome when you customize the Django admin, you’ll need some models. Edit core/models.py:

```
from django.core.validators import MinValueValidator, MaxValueValidator
from django.db import models


class Person(models.Model):
    last_name = models.TextField()
    first_name = models.TextField()
    courses = models.ManyToManyField("Course", blank=True)


    class Meta:
        verbose_name_plural = "People"

class Course(models.Model):
    name = models.TextField()
    year = models.IntegerField()

    class Meta:
        unique_together = ("name", "year", )

class Grade(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    grade = models.PositiveSmallIntegerField(
        validators=[MinValueValidator(0), MaxValueValidator(100)])
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
```

These models represent students taking courses at a school. A Course has a name and a year in which it was offered. A Person has a first and last name and can take zero or more courses. A Grade contains a percentage score that a Person received on a Course.

Here's a model diagram showing the relationships between the objects:



The underlying table names in the database are slightly different from this, but they're related to the models shown above.

Each model that you want Django to represent in the admin interface needs to be registered. You do this in the admin.py file. Models from core/models.py are registered in the corresponding core/admin.py file:

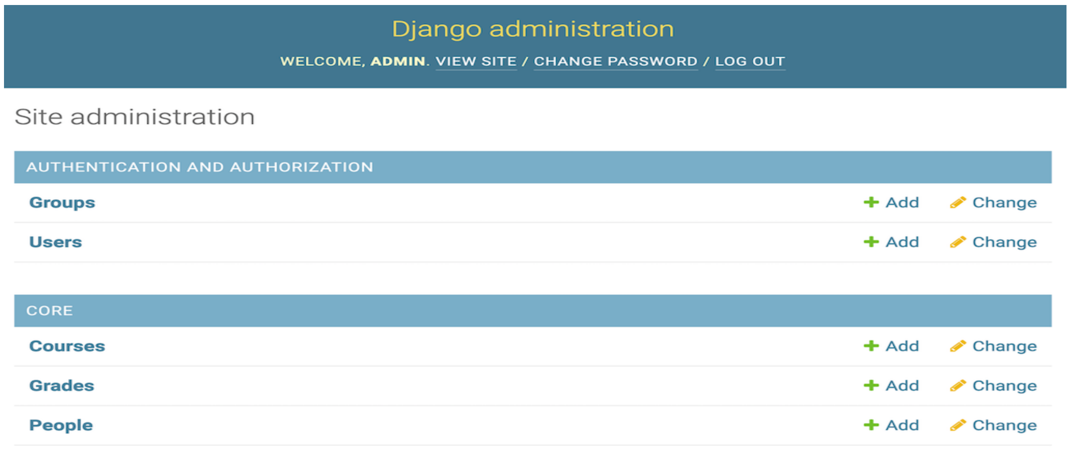
```
from django.contrib import admin


from core.models import Person, Course, Grade


@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    pass


@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    pass


@admin.register(Grade)
class GradeAdmin(admin.ModelAdmin):
    pass
```

You're almost ready to go. Once you've migrated your database models, you can run the Django development server and see the results:

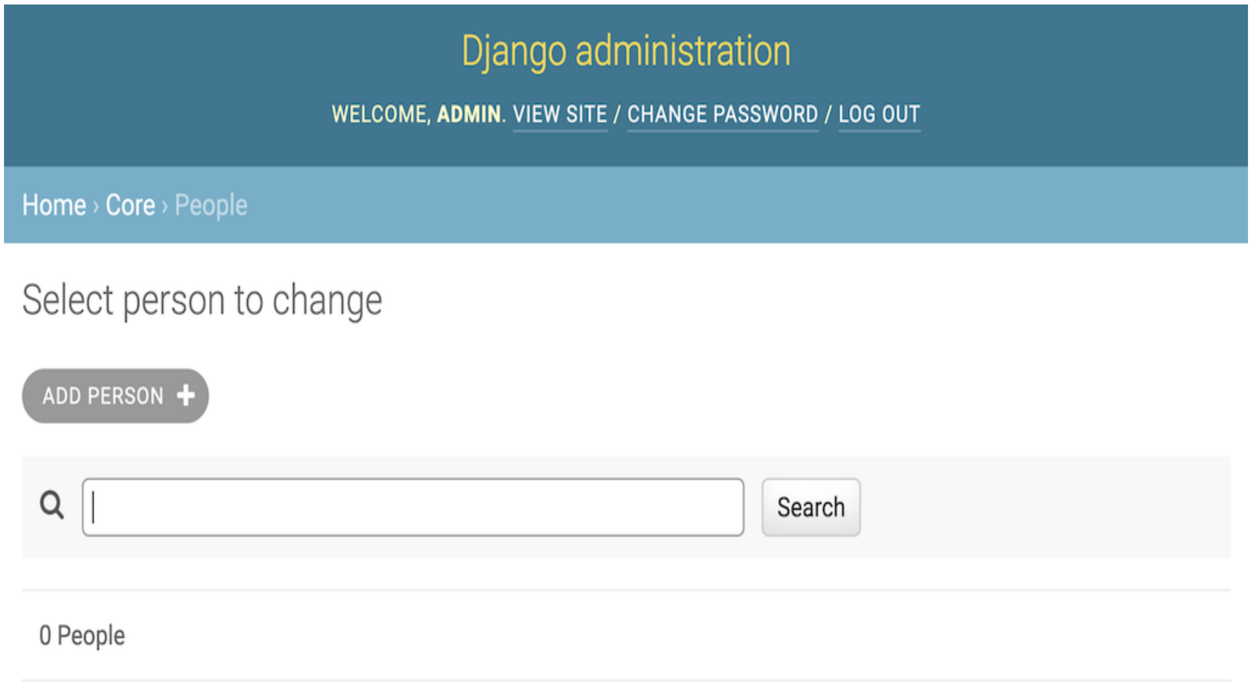
```
$ ./manage.py makemigrations
$ ./manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, core, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  ...
  Applying core.0001_initial... OK
  Applying core.0002_auto_20200609_2120... OK
  Applying sessions.0001_initial... OK
$ ./manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
Django version 3.0.7, using settings 'School.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Now visit http://127.0.0.1:8000/admin to see your admin interface. You'll be prompted to log in. Use the credentials you created with the createsuperuser management command.

The admin home screen lists all the registered database models:



You can now use the interface to create objects in your database. Clicking a model name will show you a screen listing all the objects in the database for that model. Here's the Person list:


Django administration

WELCOME, **ADMIN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Core › People

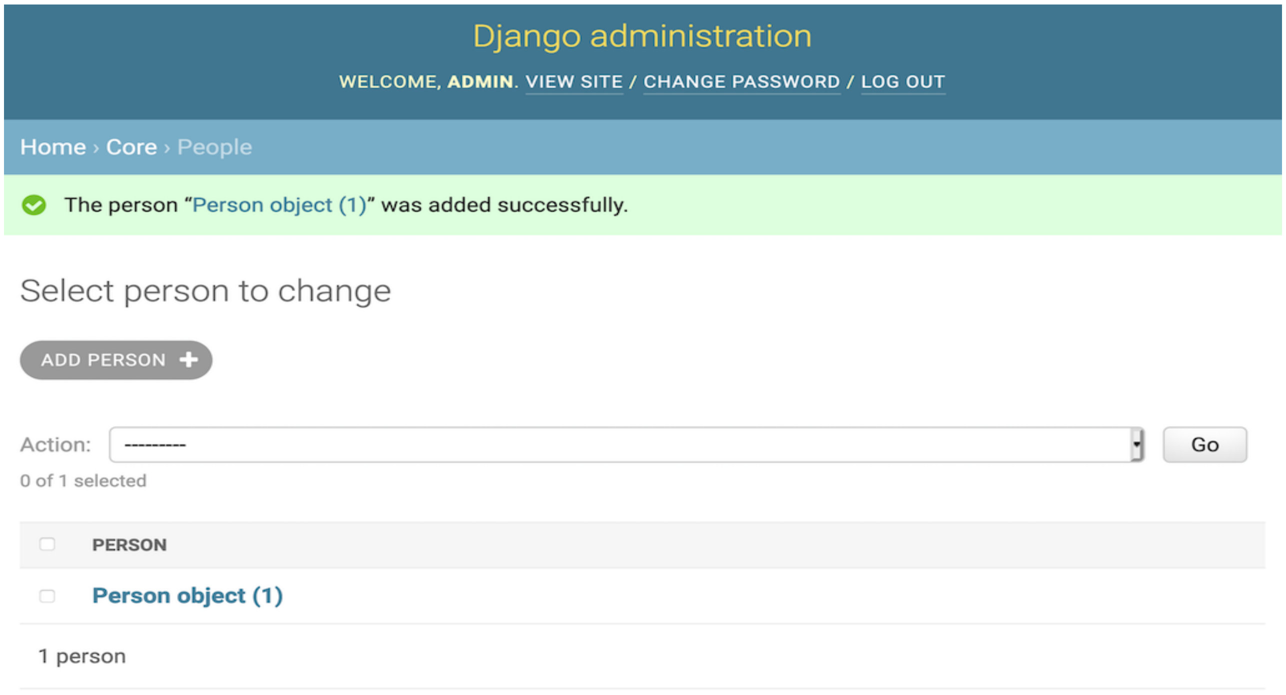
Select person to change

ADD PERSON

Search

0 People

The list starts out empty, like your database. Clicking *ADD PERSON* allows you to create a person in the database. Once you save, you'll be returned to the list of Person objects:


Django administration

WELCOME, **ADMIN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Core › People

The person "Person object (1)" was added successfully.

Select person to change

ADD PERSON

Action: --------- Go

0 of 1 selected

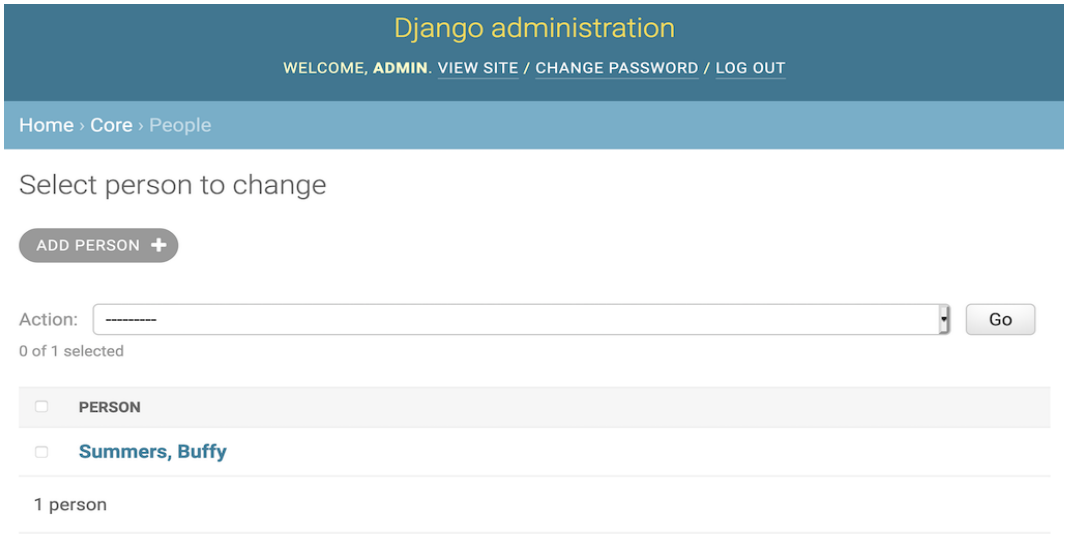
PERSON
Person object (1)

1 person

The good news is you've got an object. The bad news is Person object (1) tells you the id of the object and nothing else. By default, the Django admin displays each object by calling str() on it. You can make this screen a little more helpful by adding a .__str__() method to the Person class in core/models.py:

```
class Person(models.Model):
    last_name = models.TextField()
    first_name = models.TextField()
    courses = models.ManyToManyField("Course", blank=True)


    def __str__(self):
        return f"{self.last_name}, {self.first_name}"
```

Adding Person.__str__() changes the display to include the first and last name of the Person in the interface. You can refresh the screen to see the change:



Select person to change

0 of 1 selected

	PERSON
	Summers, Buffy

1 person

That's a little better! Now you can see some information about the Person object. It's a good idea to add similar methods to both the Course and the Grade objects:

```
class Course(models.Model):
    # ...
    def __str__(self):
        return f"{self.name}, {self.year}"

class Grade(models.Model):
    # ...

    def __str__(self):
        return f"{self.grade}, {self.person}, {self.course}"
```

You’ll want to have some data in your database to see the full effect of your customizations. You can have some fun and create your own data now, or you can skip the work and use a **fixture**. Expand the box below to learn how to load data using a fixture.

Loading Fixtures in DjangoShow/Hide

Now that you have some data to work with, you’re ready to start customizing Django’s admin interface.

Customizing the Django Admin

The smart folks who created the Django framework not only built the admin, but they did it in such a way that you can customize it for your projects. When you registered the PersonAdmin object earlier, it inherited from admin.ModelAdmin. Most of the customization you can do with the Django admin is done by modifying ModelAdmin, and you sure can modify it!

ModelAdmin has over thirty attributes and almost fifty methods. You can use each one of these to fine-tune the admin’s presentation and control your objects’ interfaces. Every one of these options is described in detail in the documentation.

To top it all off, the admin is built using Django’s templating interface. The Django template mechanism allows you to override existing templates, and because the admin is just another set of templates, this means you can completely change its HTML.

Although it’s beyond the scope of this tutorial, you can even create multiple admin sites. That might seem like overkill, but it allows you to get fancy and define different sites for users with different permissions.

The Django admin is split into three major areas:

1. App index
2. Change lists
3. Change forms

The **app index** lists your registered models. A **change list** is automatically created for each registered model and lists the objects for that model. When you add or edit one of those objects, you do so with a **change form**.

In the earlier example, the app index showed the Person, Course, and Grade objects. Clicking *People* shows the change lists for Person objects. On the Person change list page, clicking the Buffy Summers object takes you to the change form to edit Buffy’s details.

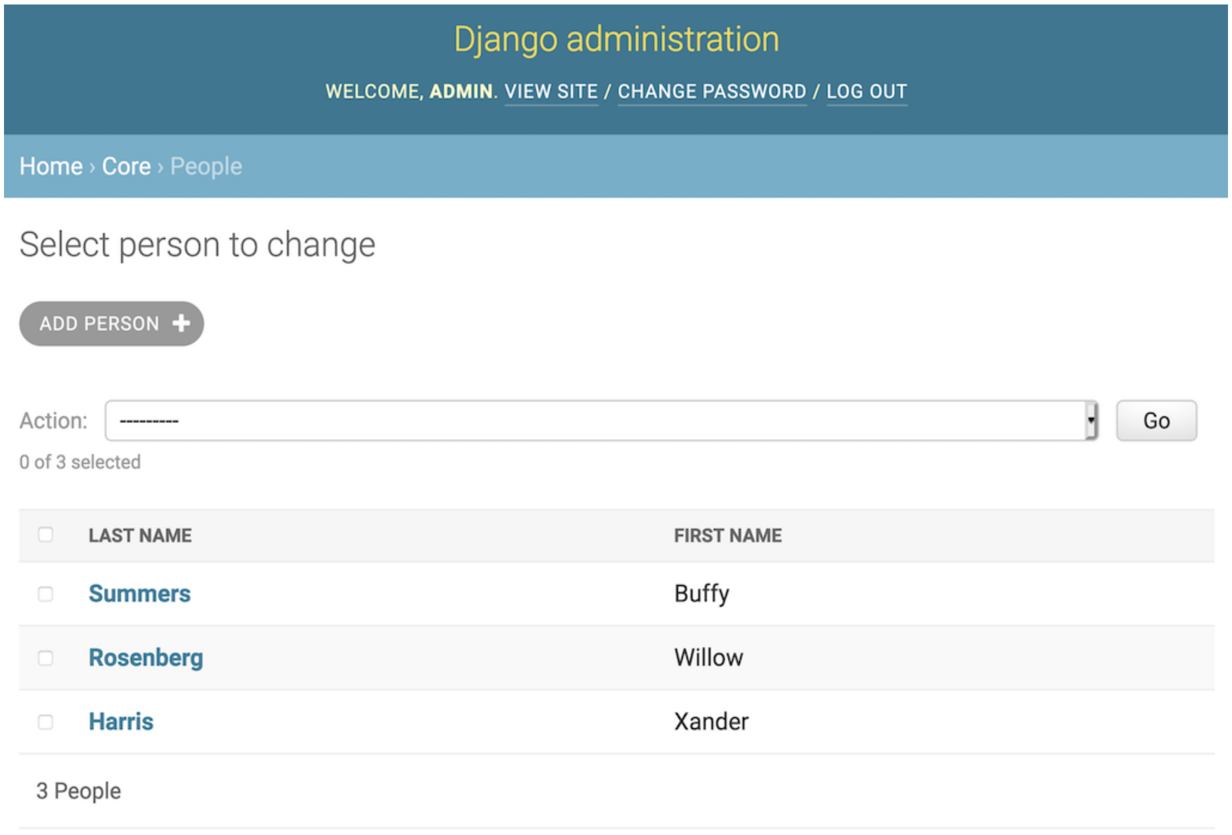
Modifying a Change List Using list_display

Implementing .__str__() is a quick way to change the representation of a Person object from a meaningless string to understandable data. Since this representation will also show up in drop-downs and multi-selects, you definitely want to make it as easy to understand as possible.

You can customize change list pages in far more ways than just modifying an object’s string representation. The list_display attribute of an admin.ModelAdmin object specifies what columns are shown in the change list. This value is a tuple of attributes of the object being modeled. For example, in core/admin.py, modify PersonAdmin as follows:

```
@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
   list_display = ("last_name", "first_name")
```

The code above modifies your Person change list to display
the last_name and first_name attributes for each Person object. Each attribute is shown in a column on the page:



Select person to change

0 of 3 selected

LAST NAME	FIRST NAME
Summers	Buffy
Rosenberg	Willow
Harris	Xander

3 People

The two columns are clickable, allowing you to sort the page by the column data. The admin also respects the ordering attribute of a Meta section:

```
class Person(models.Model):
    # ...


    class Meta:
        ordering = ("last_name", "first_name")


    # ...
```

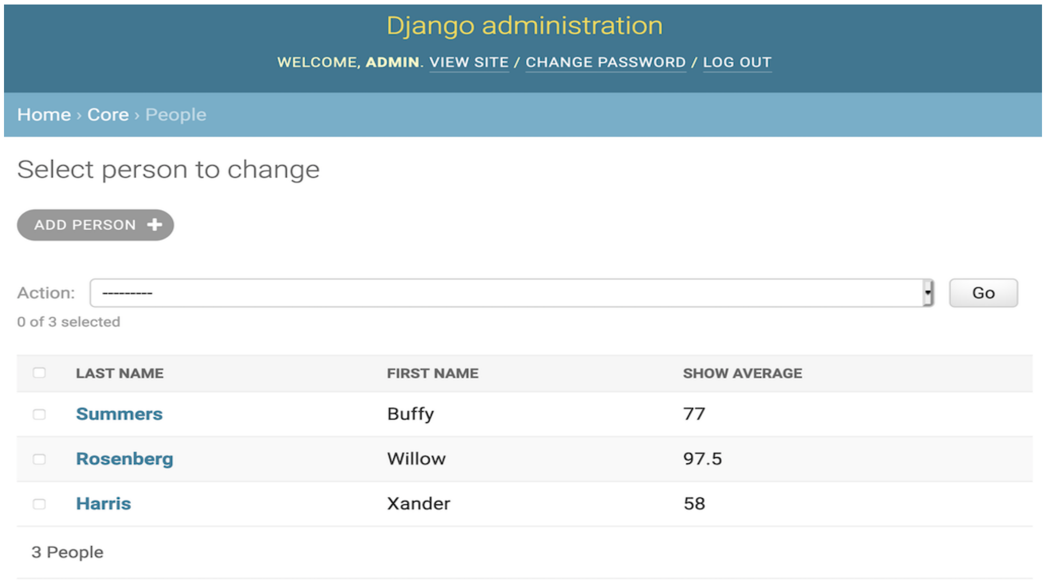
Adding the ordering attribute will default all queries on Person to be ordered by last_name then first_name. Django will respect this default order both in the admin and when fetching objects.

The list_display tuple can reference any attribute of the object being listed. It can also reference a method in the admin.ModelAdmin itself. Modify PersonAdmin again:

```
@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    list_display = ("last_name", "first_name", "show_average")

    def show_average(self, obj):
        from django.db.models import Avg
        result = Grade.objects.filter(person=obj).aggregate(Avg("grade"))
        return result["grade__avg"]
```

In the above code, you add a column to the admin that displays each student’s grade average. show_average() is called once for each object displayed in the list.

The obj parameter is the object for the row being displayed. In this case, you use it to query the corresponding Grade objects for the student, with the response averaged over Grade.grade. You can see the results here:


Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Core › People

Select person to change

ADD PERSON

Action: --------- Go

0 of 3 selected

LAST NAME	FIRST NAME	SHOW AVERAGE
Summers	Buffy	77
Rosenberg	Willow	97.5
Harris	Xander	58

3 People

Keep in mind that the average grade should really be calculated in the Person model object. You'll likely want the data elsewhere, not just in the Django admin. If you had such a method, you could add it to the list_display attribute. The example here shows what you can do in a ModelAdmin object, but it probably isn't the best choice for your code.

By default, only those columns that are object attributes are sortable. show_average() is not. This is because sorting is performed by an underlying QuerySet, not on the displayed results. There are ways of sorting these columns in some cases, but that's beyond the scope of this tutorial.

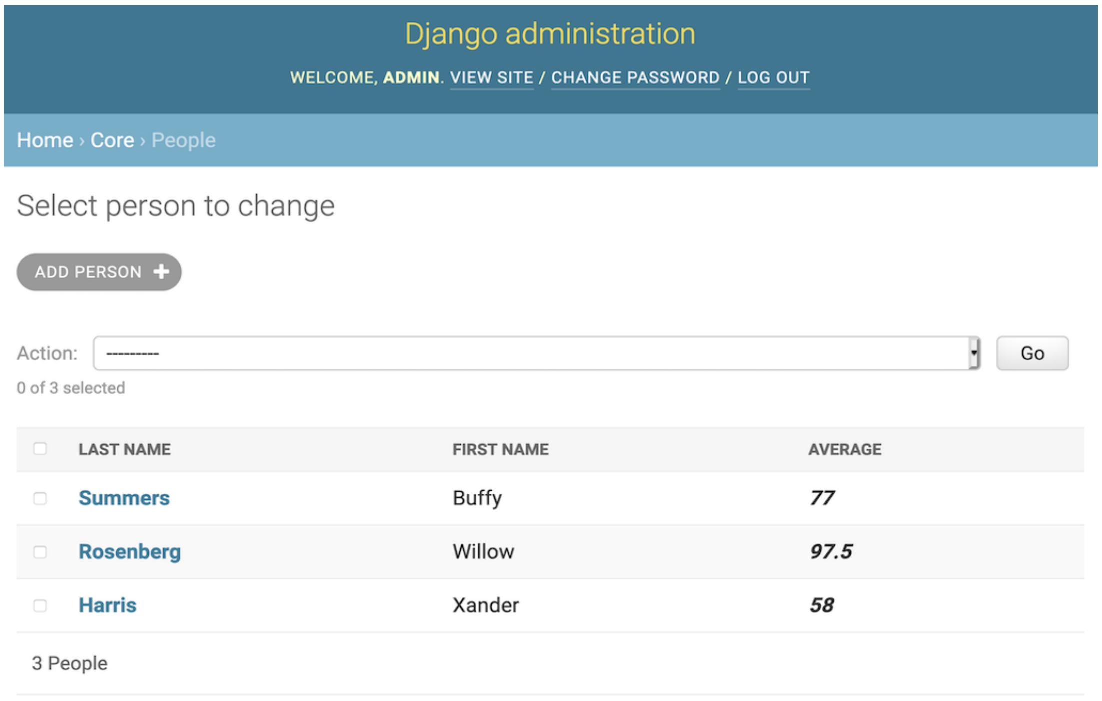
The title for the column is based on the name of the method. You can alter the title by adding an attribute to the method:

```
def show_average(self, obj):
    result = Grade.objects.filter(person=obj).aggregate(Avg("grade"))
    return result["grade__avg"]


show_average.short_description = "Average Grade"
```

By default, Django protects you from HTML in strings in case the string is from user input. To have the display include HTML, you must use format_html():

```
def show_average(self, obj):
    from django.utils.html import format_html


    result = Grade.objects.filter(person=obj).aggregate(Avg("grade"))
    return format_html("<b><i>{}</i></b>", result["grade__avg"])


show_average.short_description = "Average"
```

show_average() now has a custom title, "Average", and is formatted to be in italics:



Select person to change

0 of 3 selected

LAST NAME	FIRST NAME	AVERAGE
Summers	Buffy	***77***
Rosenberg	Willow	***97.5***
Harris	Xander	***58***

3 People

Unfortunately, Django hasn't yet added f-string support for format_html(), so you're stuck with str.format() syntax.

Providing Links to Other Object Pages

It's quite common for objects to reference other objects through the use of **foreign keys**. You can point list_display at a method that returns an HTML link. Inside core/admin.py, modify the CourseAdmin class as follows:

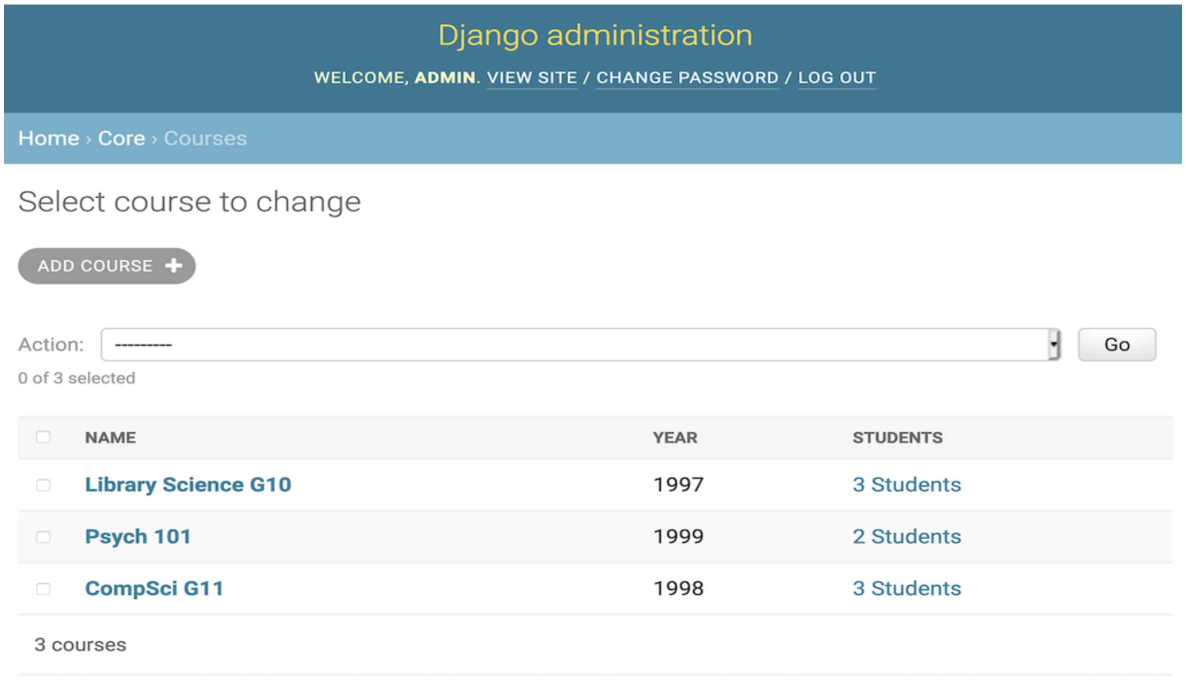
```
from django.urls import reverse
from django.utils.http import urlencode
```

```
@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ("name", "year", "view_students_link")

    def view_students_link(self, obj):
        count = obj.person_set.count()
        url = (
            reverse("admin:core_person_changelist")
            + "?"
            + urlencode({"courses__id": f"{obj.id}"})
        )
        return format_html('<a href="{}">{} Students</a>', url, count)

    view_students_link.short_description = "Students"
```

This code causes the Course change list to have three columns:

1. The course name
2. The year in which the course was offered
3. A link displaying the number of students in the course

You can see the resulting change in the following screenshot:



	NAME	YEAR	STUDENTS
	Library Science G10	1997	3 Students
	Psych 101	1999	2 Students
	CompSci G11	1998	3 Students

3 courses

When you click *2 Students*, it takes you to the Person change list page with a filter applied. The filtered page shows only those students in Psych 101, Buffy and Willow. Xander didn't make it to university.

The example code uses reverse() to look up a URL in the Django admin. You can look up any admin page using the following naming convention:

```
"admin:%(app)s_%(model)s_%(page)"
```

This name structure breaks down as follows:

- **admin:** is the namespace.
- **app** is the name of the app.
- **model** is the model object.
- **page** is the Django admin page type.

For the view_students_link() example above, you use admin:core_person_changelist to get a reference to the change list page of the Person object in the core app.

Here are the available URL names:

Page	URL Name	Purpose
Change list	%(app)s_%(model)s_changelist	Model object page list
Add	%(app)s_%(model)s_add	Object creation page
History	%(app)s_%(model)s_history	Object change history page Takes an object_id as a parameter
Delete	%(app)s_%(model)s_delete	Object delete page Takes an object_id as a parameter
Change	%(app)s_%(model)s_change	Object edit page Takes an object_id as a parameter

You can filter the change list page by adding a query string to the URL. This query string modifies the QuerySet used to populate the page. In the example above, the query string "?courses__id={obj.id}" filters the Person list to only those objects that have a matching value in Person.course.

These filters support QuerySet field lookups using double underscores (__). You can access attributes of related objects as well as use filter modifiers like __exact and __startswith.

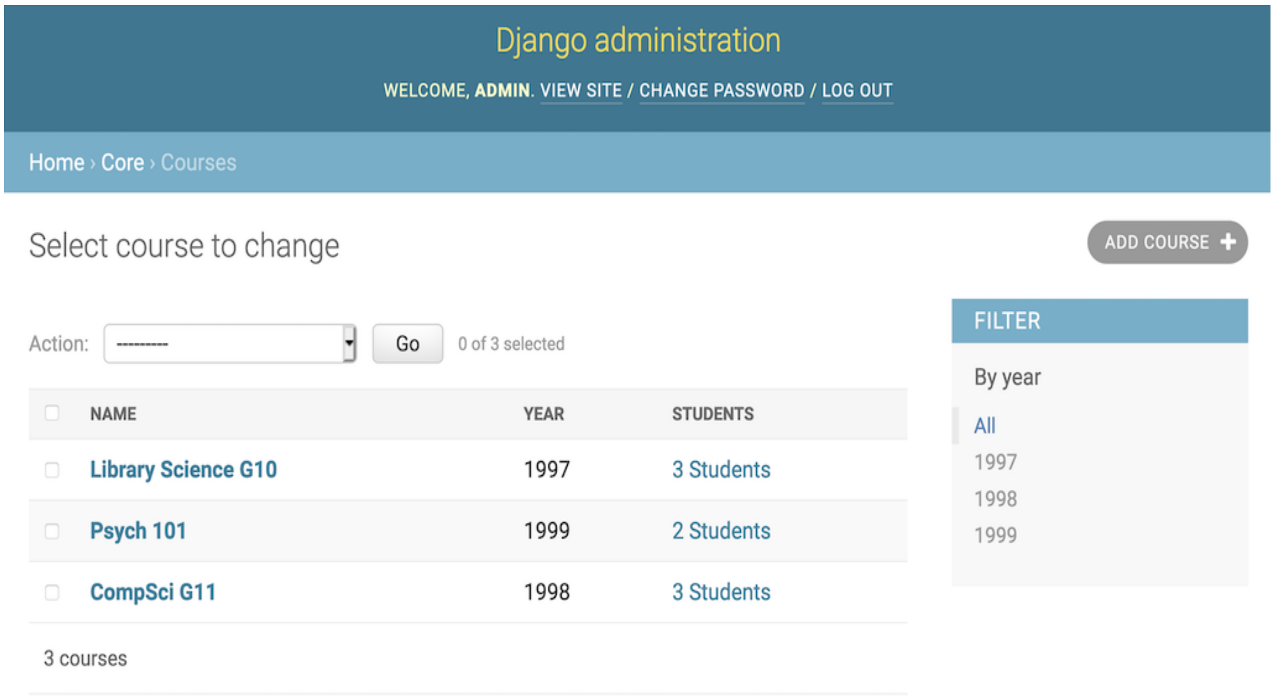
You can find the full details on what you can accomplish with the list_display attribute in the Django admin documentation.

Adding Filters to the List Screen

In addition to filtering data on the change list through the calling URL, you can also filter with a built-in widget. Add the list_filter attribute to the CourseAdmin object in core/admin.py:

```
@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ("name", "year", "view_students_link")
    list_filter = ("year", )
# ...
```

The list_filter will display a new section on the page with a list of links. In this case, the links filter the page by year. The filter list is automatically populated with the year values used by the Course objects in the database:


Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Core › Courses

Select course to change

ADD COURSE +

Action: --------- Go 0 of 3 selected

	NAME	YEAR	STUDENTS
☐	Library Science G10	1997	3 Students
☐	Psych 101	1999	2 Students
☐	CompSci G11	1998	3 Students

3 courses

FILTER

By year

All
1997
1998
1999

Clicking a year on the right-hand side will change the list to include only Course objects with that year value. You can also filter based on the attributes of related objects using the __ field lookup syntax. For example, you could filter GradeAdmin objects by course__year, showing the Grade objects for only a certain year of courses.

If you're looking for more control over your filtering, then you can even create **filter objects** that specify the lookup attributes and the corresponding QuerySet.

Adding Search to the List Screen

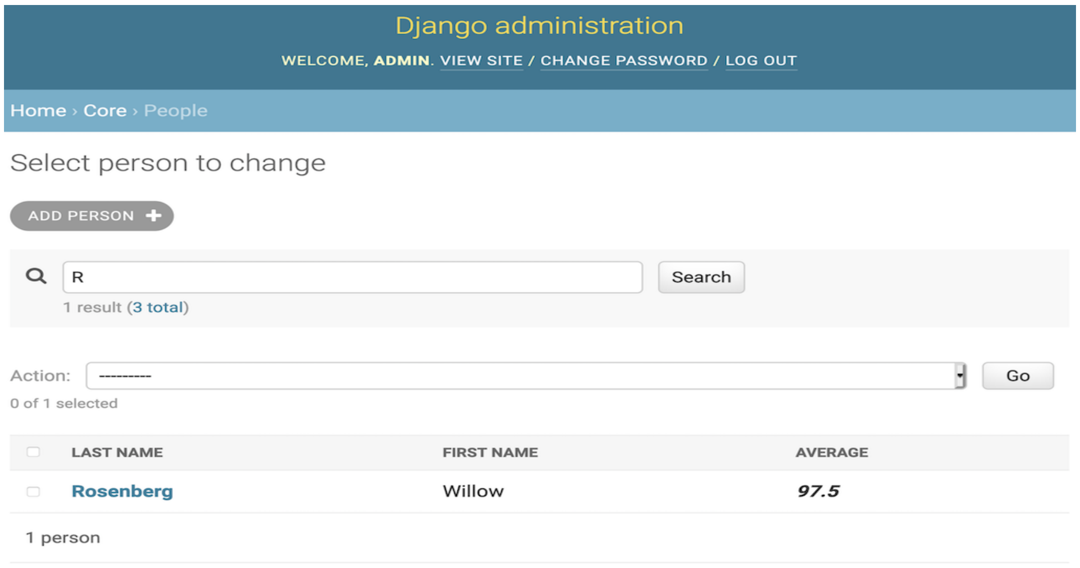
Filters aren't the only way to reduce the amount of data on the screen. Django admin also supports searching through the search_fields option, which adds a **search box** to the screen. You set it with a tuple containing the names of fields to be used for constructing a search query in the database.

Anything the user types in the search box is used in an OR clause of the fields filtering the QuerySet. By default, each search parameter is surrounded by % signs, meaning if you search for r, then any word with an r inside will appear in the results. You can be more precise by specifying a __ modifier on the search field.

Edit the PersonAdmin in core/admin.py as follows:

```
@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    search_fields = ("last_name__startswith", )
```

In the above code, searching is based on last name. The __startswith modifier restricts the search to last names that begin with the search parameter. Searching on R provides the following results:


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Core › People
Select person to change

1 result (3 total)
0 of 1 selected

LAST NAME	FIRST NAME	AVERAGE
Rosenberg	Willow	97.5

1 person

Whenever a search is performed on a change list page, the Django admin calls your admin.ModelAdmin object's get_search_results() method. It returns a QuerySet with the search results. You can fine-tune searches by overloading the method and changing the QuerySet. More details can be found in the documentation.

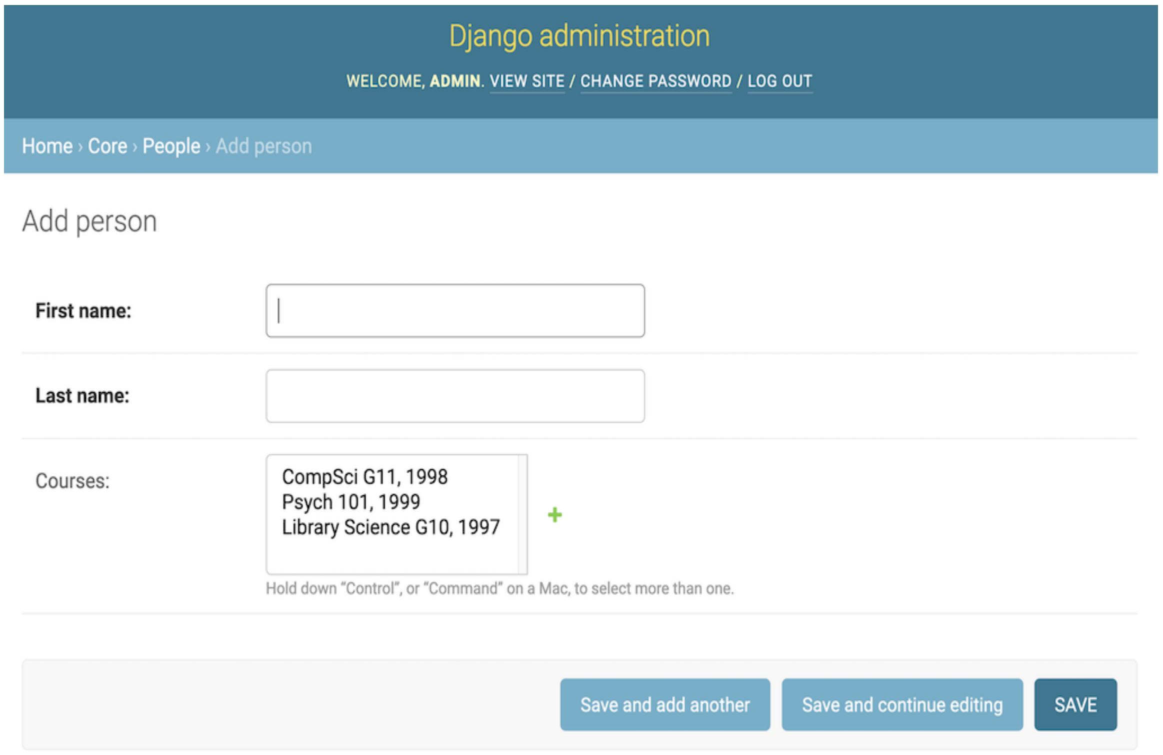
Changing How Models Are Edited

You can customize more than just the change list page. The screens used to add or change an object are based on a ModelForm. Django automatically generates the form based on the model being edited.

You can control which fields are included, as well as their order, by editing the fields option. Modify your PersonAdmin object, adding a fields attribute:

```
@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    fields = ("first_name", "last_name", "courses")
# ...
```

The Add and Change pages for Person now put the first_name attribute before the last_name attribute even though the model itself specifies the other way around:


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Core › People › Add person
Add person
First name:
Last name:
Courses:
CompSci G11, 1998
Psych 101, 1999
Library Science G10, 1997
Hold down "Control", or "Command" on a Mac, to select more than one.
Save and add another
Save and continue editing
SAVE

ModelAdmin.get_form() is responsible for creating the ModelForm for your object. You can override this method to change the form. Add the following method to PersonAdmin:

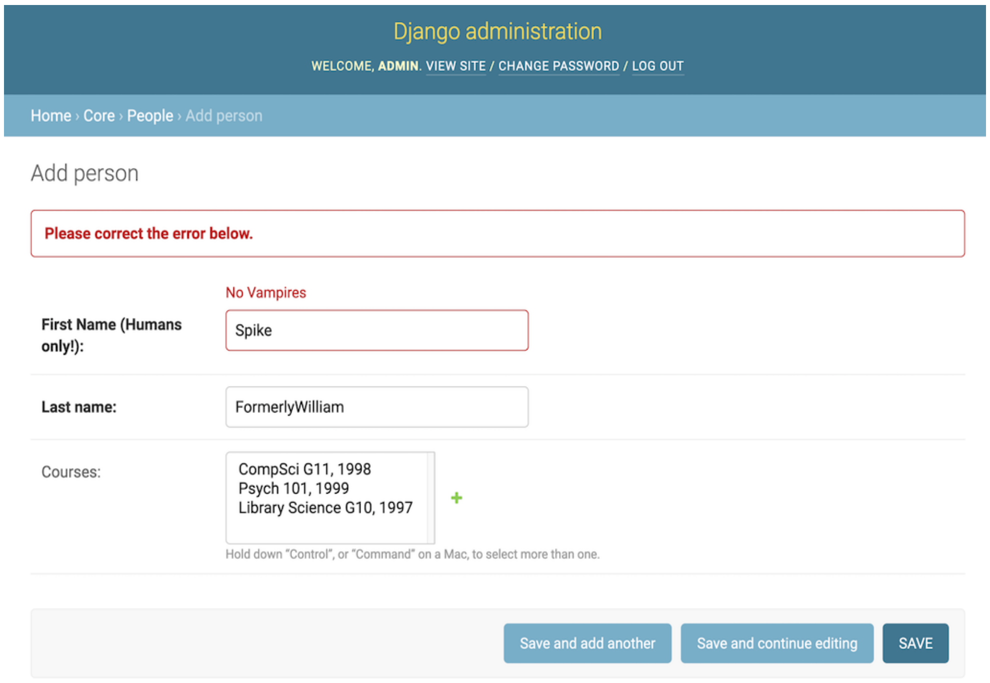
```
def get_form(self, request, obj=None, **kwargs):
    form = super().get_form(request, obj, **kwargs)
    form.base_fields["first_name"].label = "First Name (Humans only!):"
    return form
```

Now, when the Add or Change page is displayed, the label of the first_name field will be customized.

Changing the label might not be sufficient to prevent vampires from registering as students. If you don't like the ModelForm that the Django admin created for you, then you can use the form attribute to register a custom form. Make the following additions and changes to core/admin.py:

```
from django import forms


class PersonAdminForm(forms.ModelForm):
    class Meta:
        model = Person
        fields = "__all__"


    def clean_first_name(self):
        if self.cleaned_data["first_name"] == "Spike":
            raise forms.ValidationError("No Vampires")


        return self.cleaned_data["first_name"]


@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    form = PersonAdminForm
# ...
```

The above code enforces additional validation on the Person Add and Change pages. ModelForm objects have a rich validation mechanism. In this case, the first_name field is being checked against the name "Spike". A ValidationError prevents students with this name from registering:


Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Core › People › Add person

Add person

Please correct the error below.

No Vampires

First Name (Humans only!):	Spike
Last name:	FormerlyWilliam
Courses:	CompSci G11, 1998 Psych 101, 1999 Library Science G10, 1997

Hold down "Control", or "Command" on a Mac, to select more than one.

By changing or replacing the ModelForm object, you can fully control the appearance and validation of the pages you use to add or change object pages.

Overriding Django Admin Templates

The Django developers implemented the admin using the Django template mechanism. This made their job a little bit easier, but it also benefits you by allowing you to **override** the templates. You can fully customize the admin by changing the templates used to render pages.

You can see all the templates used in the admin by looking inside the Django package in your virtual environment:

```
.../site-packages/django/contrib/admin/templates/
│
├── admin/
│   │
│   ├── auth/
│   │   └── user/
│   │       ├── add_form.html
│   │       └── change_password.html
│   │
│   ├── edit_inline/
│   │   ├── stacked.html
│   │   └── tabular.html
│   │
│   ├── includes/
│   │   ├── fieldset.html
│   │   └── object_delete_summary.html
│   │
│   ├── widgets/
│   │   ├── clearable_file_input.html
│   │   ├── foreign_key_raw_id.html
│   │   ├── many_to_many_raw_id.html
│   │   ├── radio.html
│   │   ├── related_widget_wrapper.html
│   │   ├── split_datetime.html
│   │   └── url.html
│   │
│   ├── 404.html
│   ├── 500.html
│   ├── actions.html
│   ├── app_index.html
│   ├── base.html
│   ├── base_site.html
│   ├── change_form.html
│   ├── change_form_object_tools.html
│   ├── change_list.html
│   ├── change_list_object_tools.html
│   ├── change_list_results.html
│   ├── date_hierarchy.html
│   ├── delete_confirmation.html
│   ├── delete_selected_confirmation.html
│   ├── filter.html
│   ├── index.html
│   ├── invalid_setup.html
│   ├── login.html
│   ├── object_history.html
│   ├── pagination.html
│   ├── popup_response.html
│   ├── prepopulated_fields_js.html
│   ├── search_form.html
│   └── submit_line.html
│
└── registration/
    ├── logged_out.html
    ├── password_change_done.html
    ├── password_change_form.html
    ├── password_reset_complete.html
    ├── password_reset_confirm.html
    ├── password_reset_done.html
    ├── password_reset_email.html
    └── password_reset_form.html
```

The Django template engine has a defined order for loading templates. When it loads a template, it uses the first template that matches the name. You can override admin templates by using the same directory structure and file names.

The admin templates come in two directories:

1. **admin** is for the model object pages.
2. **registration** is for password changes and logging in and out.

To customize the logout page, you need to override the right file. The relative path leading to the file has to be the same as the one being overridden. The file you're interested in is registration/logged_out.html. Start by creating the directory in the School project:

```
$ mkdir -p templates/registration
```

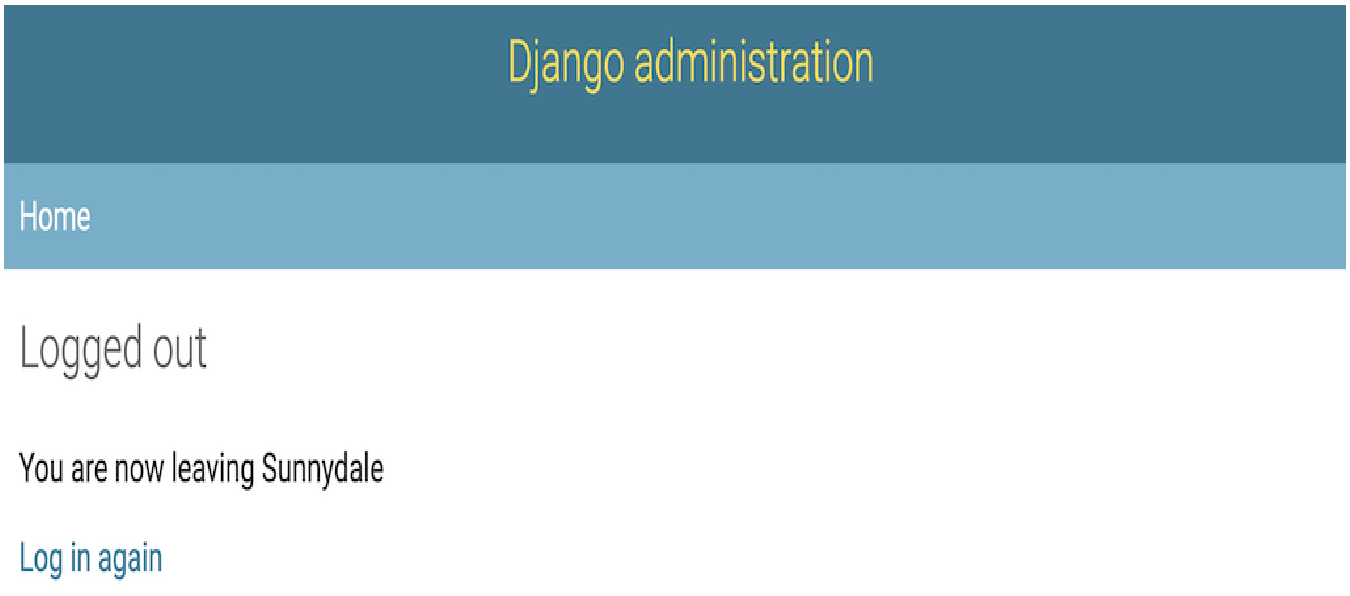
Now tell Django about your new template directory inside your School/settings.py file. Look for the TEMPLATES directive and add the folder to the DIR list:

```
# School/settings.py
# ...

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",

        # Add the templates directory to the DIR option:
        "DIRS": [os.path.join(BASE_DIR, "templates"), ],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]
```

The template engine searches directories in the DIR option before the application directories, so anything with the same name as an admin template will be loaded instead. To see this in action, copy the logged_out.html file into your templates/registration directory, then modify it:

```
{% extends "admin/base_site.html" %}
{% load i18n %}

{% block breadcrumbs %}<div class="breadcrumbs"><a href="{% url 'admin:index' %}">{%
trans 'Home' %}</a></div>{% endblock %}

{% block content %}

<p>You are now leaving Sunnydale</p>

<p><a href="{% url 'admin:index' %}">{% trans 'Log in again' %}</a></p>

{% endblock %}
```

You’ve now customized the logout page. If you click *LOG OUT*, then you’ll see the customized message:



Logged out

You are now leaving Sunnydale

Django admin templates are deeply nested and not very intuitive, but you have full control over their presentation if you need it. Some packages, including Grappelli and Django Admin Bootstrap, have fully replaced the Django admin templates to change their appearance.

Django Admin Bootstrap is not yet compatible with Django 3, and Grappelli only recently added support, so it may still have some issues. That being said, if you want to see the power of overriding admin templates, then check out those projects!

The Django admin is a powerful built-in tool giving you the ability to create, update, and delete objects in your database using a web interface. You can customize the Django admin to do almost anything you want.

Django QuerySet

Django QuerySet

A QuerySet is a collection of data from a database.

A QuerySet is built up as a list of objects.

QuerySets makes it easier to get the data you actually need, by allowing you to filter and order the data at an early stage.

In this tutorial we will be querying data from the Member table.

Member:

id	firstname	lastname	phone	joined_date
1	Emil	Refsnes	5551234	2022-01-05
2	Tobias	Refsnes	5557777	2022-04-01
3	Linus	Refsnes	5554321	2021-12-24

4	Lene	Refsnes	5551234	2021-05-01
5	Stalikken	Refsnes	5559876	2022-09-29

Querying Data

In views.py, we have a view for testing called testing where we will test different queries.

In the example below we use the .all() method to get all the records and fields of the Member model:

ViewGet your own Django Server

views.py:

```
from django.http import HttpResponse

from django.template import loader

from .models import Member



def testing(request):

  mydata = Member.objects.all()

  template = loader.get_template('template.html')

  context = {

    'mymembers': mydata,

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

The object is placed in a variable called mydata, and is sent to the template via the context object as mymembers, and looks like this:

```
<QuerySet [
  <Member: Member object (1)>,
  <Member: Member object (2)>,
  <Member: Member object (3)>,
  <Member: Member object (4)>,
  <Member: Member object (5)>
]>
```

As you can see, our Member model contains 5 records, and are listed inside the QuerySet as 5 objects.

In the template you can use the mymembers object to generate content:

```
Template

templates/template.html:
```

```
<table border='1'>

  <tr>

    <th>ID</th>

    <th>Firstname</th>

    <th>Lastname</th>

  </tr>

  {% for x in mymembers %}
  <tr>
    <td>{{ x.id }}</td>
      <td>{{ x.firstname }}</td>
    <td>{{ x.lastname }}</td>
  </tr>
 {% endfor %}
</table>
```

ID	**Firstname**	**Lastname**
1	Emil	Refsnes
2	Tobias	Refsnes
3	Linus	Refsnes
4	Lene	Refsnes
5	Stalikken	Refsnes

In views.py you can see how to import and fetch members from the database.

Building list and detail views

Now that you have knowledge of how to use the ORM, you are ready to build the views of the blog application. A Django view is just a Python function that receives a web request and returns a web response. All the logic to return the desired response goes inside the view.

First, you will create your application views, then you will define a URL pattern for each view, and finally, you will create HTML templates to render the data generated by the views. Each view will render a template, passing variables to it, and will return an HTTP response with the rendered output.

Creating list and detail views

Let's start by creating a view to display the list of posts. Edit the views.py file of your blog application and make it look like this:

```
from django.shortcuts import render, get_object_or_404

from .models import Post

def post_list(request):

    posts = Post.published.all()

    return render(request,

                  'blog/post/list.html',

                  {'posts': posts})
```

Copy

You just created your first Django view. The post_list view takes the request object as the only parameter. This parameter is required by all views. In this view, you retrieve all the posts with the published status using the published manager that you created previously.

Finally, you use the render() shortcut provided by Django to render the list of posts with the given template. This function takes the request object, the template path, and the context variables to render the given template. It returns an HttpResponse object with the rendered text (normally HTML code). The render() shortcut takes the request context into account, so any variable set by the template context processors is accessible by the given template. Template context processors are just callables that set variables into the context. You will learn how to use them in *Chapter 3, Extending Your Blog Application*.

Let's create a second view to display a single post. Add the following function to the views.py file:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post,
                                   status='published',
                                   publish__year=year,
                                   publish__month=month,
                                   publish__day=day)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

Copy

This is the post detail view. This view takes the year, month, day, and post arguments to retrieve a published post with the given slug and date. Note that when you created the Post model, you added the unique_for_date parameter to the slug field. This ensures that there will be only one post with a slug for a given date, and thus, you can retrieve single posts using the date and slug. In the detail view, you use the get_object_or_404() shortcut to retrieve the desired post. This function retrieves the object that matches the given parameters or an HTTP 404 (not found) exception if no object is found. Finally, you use the render() shortcut to render the retrieved post using a template.

Adding URL patterns for your views

URL patterns allow you to map URLs to views. A URL pattern is composed of a string pattern, a view, and, optionally, a name that allows you to name the URL project-wide. Django runs through each URL pattern and stops at the first one that matches the requested URL. Then, Django imports the view of the matching URL pattern and executes it, passing an instance of the HttpRequest class and the keyword or positional arguments.

Create a urls.py file in the directory of the blog application and add the following lines to it:

```
from django.urls import path

from . import views

app_name = 'blog'

urlpatterns = [

    # post views

    path('', views.post_list, name='post_list'),

    path('<int:year>/<int:month>/<int:day>/<slug:post>/',

         views.post_detail,

         name='post_detail'),

]
```

Copy

In the preceding code, you define an application namespace with the app_name variable. This allows you to organize URLs by application and use the name when referring to them. You define two different patterns using the path() function. The first URL pattern doesn't take any arguments and is mapped to the post_list view. The second pattern takes the following four arguments and is mapped to the post_detail view:

- year: Requires an integer
- month: Requires an integer
- day: Requires an integer
- post: Can be composed of words and hyphens

You use angle brackets to capture the values from the URL. Any value specified in the URL pattern as <parameter> is captured as a string. You use path converters, such as <int:year>, to specifically match and return an integer and <slug:post> to specifically match a slug. You can see all path converters provided by Django at https://docs.djangoproject.com/en/3.0/topics/http/urls/#path-converters.

If using path() and converters isn't sufficient for you, you can use re_path() instead to define complex URL patterns with Python regular expressions. You can learn more about defining URL patterns with regular expressions at https://docs.djangoproject.com/en/3.0/ref/urls/#django.urls.re_path. If you haven't worked with regular expressions before, you might want to take a look at the *Regular Expression HOWTO* located at https://docs.python.org/3/howto/regex.html first.

Creating a urls.py file for each application is the best way to make your applications reusable by other projects.

Next, you have to include the URL patterns of the blog application in the main URL patterns of the project.

Edit the urls.py file located in the mysite directory of your project and make it look like the following:

```
from django.urls import path, include

from django.contrib import admin
```

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

Copy

The new URL pattern defined with include refers to the URL patterns defined in the blog application so that they are included under the blog/ path. You include these patterns under the namespace blog. Namespaces have to be unique across your entire project. Later, you will refer to your blog URLs easily by using the namespace followed by a colon and the URL name, for example, blog:post_list and blog:post_detail. You can learn more about URL namespaces at https://docs.djangoproject.com/en/3.0/topics/http/urls/#url-namespaces.

Canonical URLs for models

A canonical URL is the preferred URL for a resource. You may have different pages in your site where you display posts, but there is a single URL that you use as the main URL for a blog post. The convention in Django is to add a get_absolute_url() method to the model that returns the canonical URL for the object.

You can use the post_detail URL that you have defined in the preceding section to build the canonical URL for Post objects. For this method, you will use the reverse() method, which allows you to build URLs by their name and pass optional parameters. You can learn more about the URLs utility functions at https://docs.djangoproject.com/en/3.0/ref/urlresolvers/.

Edit the models.py file of the blog application and add the following code:

```
from django.urls import reverse
```

```
class Post(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('blog:post_detail',
                       args=[self.publish.year,
                             self.publish.month,
                             self.publish.day, self.slug])
```

Copy

You will use the get_absolute_url() method in your templates to link to specific posts.