

Django Tutorial (Windows 10)

Hands-on Learning Approach

Greetings and welcome to our [Django](#) tutorial. This tutorial is merely a simplified take on the official tutorial from [Django's](#) website (feel free to take a look!) and as such will not dive too deeply into the technical side of things. However, there will be some differences solely for the sake of simplicity and the user's convenience. Furthermore as this is a tutorial for Windows 10, there may be some slight differences for those who use different operating systems. Regardless, here are our recommendations for utilising this document:

- Your learning journey will primarily involve hands-on practice, rather than repetitive memorization. The more you code, the more proficient you will become!
- Please note that the content covered here may not encompass every aspect of Django. It's unrealistic to condense the entirety of Django into a single resource like this. Feel free to explore additional resources online to discover more tips and techniques.
- Keep in mind to enjoy the journey!

Table of Contents

Font Usage.....	3
Setting up a virtual environment using Anaconda Navigator.....	3
Creating your first virtual environment in Anaconda.....	3
Using our virtual environment in Visual Studio Code.....	4
Installing Django.....	4
Mysite project.....	5
Deploying our web server.....	6
Port number.....	7
Polls App (Hello World).....	7
Advanced Polls App.....	8

Font Usage

Words written in monospace are commands to be typed, or references to specific objects like script names or environments. For example:

```
def hello()
```

Terminal output will be `grey out monospace`. For example:

```
ls /  
bin  dev  home  lib64  mnt  proc  run  srv  tmp  var  
boot  etc  lib   media  opt  root  sbin  sys  usr
```

Setting up a virtual environment using Anaconda Navigator

In order to run Django we must first set up a virtual environment, which is akin to a separate area in which we can install whatever programs and mess with them however we like without interfering with the rest of our system. If you looked through the installation guide on Django's website, you'd notice that there's not mention of Anaconda whatsoever and that they created a virtual environment without it. This method of creating a virtual environment is entirely valid but the reason we use Anaconda is because it's extremely convenient, overly so in fact. To find out why or how, let us first create our own virtual environment.

Creating your first virtual environment in Anaconda

1. Launch Anaconda Navigator
2. Click on the Environment tab, and click on the Create button.
3. Name our environment `test_project`, and make sure the box for Python is checked. For this tutorial, we'll be using the latest version of Python available (3.12.3 as of the time of writing). Apply your settings, click Create and wait for the environment to be generated.

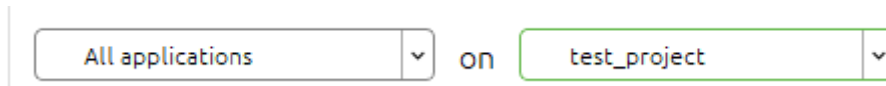
Once it's done loading, you'll see your newly-created environment on display. Right now, you should be looking at a list of items with names, descriptions and version numbers. These items are called packages and they're essentially a collection of different tools all with different purposes; Django is one such package. The reason Anaconda is convenient is because it allows us to manage the numerous different packages that we have in each of our environments. Though the effect isn't as profound when working with a single environment, as you create more and more environments you'll come to realise how much

of a life savior Anaconda is. As opposed to manually checking each environment with command lines and written notes, you could simply skip all that with a few clicks!

Using our virtual environment in Visual Studio Code

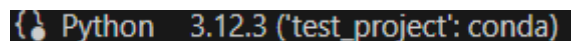
Now that we've made our virtual environment, let's proceed by learning how to use said environment in Visual Studio Code. Navigate back to the Home page and look beside it.

You will see this:



If you don't already see "test_project", click on that box and select "test_project" then allow it to load. Now, find VS Code and click Launch beneath it. This will open a Visual Studio Code that's currently using your virtual environment.

Upon opening Visual Studio Code, make a new file and change the programming language to Python. Once you've done that, your virtual environment will appear beside the language mode as such:

A screenshot of the Visual Studio Code interface showing the Python language mode. The text 'Python 3.12.3 ('test_project': conda)' is displayed in a dark background with a light border.

This shows that you're working in your virtual environment.

Installing Django

Before we proceed, there're a few pieces of information that you should know beforehand. For this project, we'll need to install Django twice. Not for fun, no. Obviously, we need to install Django but why twice? I could tell you now, but I think you'd understand better if you experienced it first hand.

To install Django, we need an updated version of "pip", a package installer for Python that comes pre-installed with Python itself. To update it, open a new Powershell terminal **using** VS Code and run:

```
py -m pip install --upgrade pip
```

Doing so will update pip if you don't already have the latest version. Now the next step is to install Django, proceed to the terminal and run:

```
py -m pip install Django
```

An especially crucial thing to note is that when it comes to code, punctuation, spelling, spacing and capitalisation are all important since any part that isn't the same may cause the code to malfunction. Upon installing Django, you may verify if it's installed by running:

```
py -m django --version
```

which will display the version of Django that's installed (only works if it's actually installed).

Mysite project

Now, let's create our project. Run:

```
django-admin startproject mysite
```

in your terminal and then find out where your project was created. For me, it's located in C:\Users\PSC. To make navigating between our various files simpler, open VS Code's explorer in the top left corner and then open the folder your project is in. If a prompt appears asking you if you trust the authors of the file then just click yes, you *are* the author after all. What you've just done is open a workspace, which not only allows you to navigate through your project's files with ease, but also to create files of different types in specific folders with a single click. If you expanded the inner mysite directory, you'd see a list of Python files all with different names; Here's what they do:

- `manage.py`: This file is a command-line utility that allows you to interact with your project in various ways including checking the changes that've been made to your website or to run your web server. We'll be using this the most in this tutorial.
- `__init__.py`: This file tells Python that the inner mysite directory should be considered a Python package.
- `settings.py`: This file acts as the settings or configuration for your project and can be modified as needed.
- `urls.py`: This file contains the URLs that your project will use and may also be reused in other projects if you wish.
- `asgi.py`: This file acts as an entry point for ASGI-compatible web servers to host your project.
- `wsgi.py`: This file acts as an entry point for WSGI-compatible web servers to host your project.

Before testing whether our web server works, make sure your VS Code terminal is set to run inside your project's folder, not the folder outside it. To do this, simply run:

```
cd
```

followed by the file path of your project. So for me, it'll look like:

```
PS C:\Users\PSC> cd C:\Users\PSC\test_project
PS C:\Users\PSC\test_project>
```

If we don't do this, the terminal won't be able to recognise any command that we run using our project's Python files. To know why, let me give an example. In my system, C:\Users\

PSC consists only of folders, and in those folders there are both files and folders. If I were to run a command like:

```
py manage.py runserver
```

(which we'll do shortly) in the terminal while it's in the PSC folder, it wouldn't work since it doesn't see the manage.py program. This command first invokes the Python interpreter so that the following parts are run using Python. It is then told to locate manage.py and carry out 'runserver' using the Python file. Although manage.py is indeed stored in the PSC folder, it is actually located one folder/layer under, so the terminal can't find it. This is why we need to change the location in which we carry out our terminal commands.

Deploying our web server

Once we've 'moved' our terminal to our project folder, run:

```
py manage.py runserver
```

in the terminal. To your surprise, you'd see that it didn't work. Instead you should see something like this:

```
Traceback (most recent call last):
  File "C:\Users\PSC\testing\manage.py", line 11, in main
...
raise ImportError(
ImportError: Couldn't import Django. Are you sure it's installed and
available on your PYTHONPATH environment variable? Did you forget to
activate a virtual environment?
```

This here is precisely why we need to install Django twice. You see, earlier when we installed Django, we actually installed it into our main system. However, the fact that we're using a virtual environment (which has different apps and packages from the rest of the system) means that this virtual environment didn't come with Django installed. Now if you wanted to, you could add Django to your environment by installing it from Anaconda. *However*, Anaconda's version of Django is not up-to-date with what we require for this tutorial, so we'll be installing it manually. Run:

```
pip install Django
```

in the terminal and wait until it says:

```
Successfully installed Django-...
```

Now try the 'runserver' command shown above again and see what happens, once it works you can either click the link given in the terminal or use this (<http://127.0.0.1:8000/>) instead. If you see a rocket and "The install worked successfully! Congratulations!" then it

means your web server was successfully deployed. To quit the server, simply press “CTRL + C” at the same time.

Port number

You may have noticed when deploying your server that the number 8000 appeared at the end of the link, this number is your server’s port number and it can be modified (if you want). To modify it, simply run the ‘runserver’ command but add any 4 digit number at the end like such:

```
py manage.py runserver 9000
```

and the link will now be <http://127.0.0.1:9000/>

Polls App (Hello World)

Now that our web server works, let’s create our “Polls” app. Run:

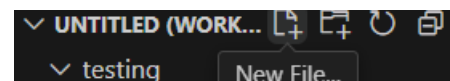
```
py manage.py startapp polls
```

in the terminal and the polls directory should show up in the workspace. Open polls/views.py (which means the views.py file in the polls folder) and edit the code inside such that it looks like this (*remember to always save files whenever you make changes to them!*):

```
from django.shortcuts import HttpResponseRedirect

def index(request):
    return HttpResponseRedirect("Hello world! You're at the polls index!")
```

Here, we’ve just created a view that when viewed on a browser will simply say “Hello World! You’re at the polls index!” in the page. However, if you deployed the server and tried visiting the page you’d simply encounter the same rocket page; This is because we haven’t told our project to find and display this view yet. To do so, create a new file named “urls.py” in the polls folder by clicking on the polls directory and then “New file”, where you can then type out urls.py:



Once you’re inside polls/urls.py, type out the following code:

```
from django.urls import path

from . import views

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

```
]
```

and then open `mysite/urls.py` and edit the code such that it looks like this:

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

urlpatterns = [
    path("polls/", include("polls.urls")),
    path("admin/", admin.site.urls),
]
```

You can leave the orange text above alone since they don't actually do anything in the code (they're simply notes). What we've just done is essentially tell our project to navigate through a specific file path and then display the view we made. Now we'll be able to see the fruit of our results; Run:

```
py manage.py runserver
```

and instead of visiting the same link (doing so will give error 404) visit <http://localhost:8000/polls/> or <http://localhost:8000/> in case either one doesn't work (change the port number in the link if you've used a different one). The page should now display the "Hello World" text without any background, which means you've succeeded.

Advanced Polls App

By default, Django uses SQLite as its database since it comes pre-installed with Python. In the future, you may want to switch to a more scalable database like PostgreSQL to avoid database-switching headaches down the road. Now, open '`mysite/settings.py`' and scroll down until line 33 where you can see the '`INSTALLED_APPS`' section. We'll go through briefly what each of these apps do:

- `django.contrib.admin`: The admin site. We'll be using this in the tutorial.
- `django.contrib.auth`: An authentication system
- `django.contrib.contenttypes`: A framework for different content types
- `django.contrib.sessions`: A session framework
- `django.contrib.messages`: A messaging framework
- `django.contrib.staticfiles`: A framework for managing static files

While running the '`runserver`' command, you may have noticed a red text regarding unapplied migrations; We'll be dealing with this now. Run:

```
py manage.py migrate
```


in the terminal and you should see something like this:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
...
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sessions.0001_initial... OK
```

Doing this makes it so that we don't encounter any annoying errors along the way.

In our Polls app we'll create 2 different models: "Question" and "Choice". "Question" will have a question and a publication date while "Choice" will have a text of choice and a vote tally. Each choice will then be associated with a question. Open polls/models.py and edit the code as such:

```
from django.db import models

class Question(models.Model):
    question_text=models.CharField(max_length=200)
    pub_date=models.DateTimeField("date published")

class Choice(models.Model):
    question=models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text=models.CharField(max_length=200)
    votes=models.IntegerField(default=0)
```

Here, each model (Question and Choice) are represented by subclasses (models.Model). Each model has a number of class variables, each of which represents a database field in the model. Each field is then represented by an instance of a 'Field' class, e.g. "CharField" for character fields and "DateTimeField" for datetimes. A relationship is then defined between the 2 models using "ForeignKey", which tells Django that each choice is related to a single question.

Now, we need to inform our project that our "Polls" app has been installed. To do this, open mysite/settings.py and edit the 'INSTALLED_APPS' section by adding:

```
"polls.apps.PollsConfig",
```

inside. It should now look like this:

```
INSTALLED_APPS = [
    "polls.apps.PollsConfig",
```

```

        "django.contrib.admin",
        "django.contrib.auth",
        "django.contrib.contenttypes",
        "django.contrib.sessions",
        "django.contrib.messages",
        "django.contrib.staticfiles",
    ]

```

To apply this change, we'll need to run:

```
py manage.py makemigrations polls
```

in the terminal and we should see:

```

Migrations for 'polls':
  polls\migrations\0001_initial.py
    - Create model Question
    - Create model Choice

```

By running 'makemigrations', we're telling Django that we've made some changes to our models (or made new models) and that you'd like these changes to be stored as a migration. Migrations are how Django stores changes to your models, they're files on disk. Now, run:

```
py manage.py sqlmigrate polls 0001
```

in the terminal to see what SQL Django thinks is required (it doesn't actually migrate anything). Note that the output will vary depending on what database you're using:

```

BEGIN;
--
-- Create model Question
--
CREATE TABLE "polls_question" ("id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT, "question_text" varchar(200) NOT NULL, "pub_date" datetime
NOT NULL);
--
-- Create model Choice
--
CREATE TABLE "polls_choice" ("id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT, "choice_text" varchar(200) NOT NULL, "votes" integer NOT
NULL, "question_id" bigint NOT NULL REFERENCES "polls_question" ("id")
DEFERRABLE INITIALLY DEFERRED);

```

```
CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice"
("question_id");

COMMIT;
```

Now run:

```
py manage.py migrate
```

in the terminal. The ‘migrate’ command checks all the unapplied migrations, runs them against your database and synchronises the changes you made to the models with schema in your database. Migrations are very powerful since they allow you to change your models over time without the need to delete your databases/tables or make new ones—it specialises in upgrading your database live, without losing any data. To make future changes to your models, remember this three step process:

1. Change your models in models.py
2. Run “py manage.py make migrations” to create migrations for those changes
3. Run “py manage.py migrate” to apply those changes to the database

Right now, we’ll be using the Python interpreter inside the terminal to generate some questions and choices for our project. Run:

```
py manage.py shell
```

in the terminal and you should now be using Python. Read the following command line carefully and enter them in order (the output will be greyed out but additional notes will begin with a hashtag #):

```
>>> from polls.models import Choice, Question
# Import the model classes we just wrote
# No Questions are in the system yet
>>> Question.objects.all()
<QuerySet []>

# Create a new Question. Support for time zones is enabled in the default
settings file, so Django expects a datetime with tzinfo for pub_date. Use
timezone.now() instead of datetime.datetime.now() and it will do the right
thing.
>>> from django.utils import timezone
>>> q=Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> q.save()
```

```

# Now it has an ID.
>>>q.id
1

# Access model field values via Python attributes.
>>>q.question_text
"What's new?"
>>>q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217,
tzinfo=datetime.timezone.utc)

# Change values by changing the attributes, then calling save().
>>> q.question_text="What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>

```

To exit the Python shell simply type:

```
exit()
```

and enter. Now let's modify our models; Open polls/models.py and edit the code as such (the added code should be placed under the code for each model):

```

from django.db import models

class Question(models.Model):
    #...
    def __str__(self):
        return self.question_text

class Choice(models.Model):
    #...
    def __str__(self):
        return self.choice_text

```

The `__str__()` method makes it more convenient when using the Python shell. Now edit `polls/models.py` again with the following (the last 2 lines are actually all in 1 line):

```
import datetime

from django.db import models
from django.utils import timezone

class Question(models.Model):
    def was_published_recently(self):
        return self.pub_date >= timezone.now() -
datetime.timedelta(days=1)
```

Make sure you only apply this change to the Question model, not the Choice model. The addition of “import datetime” allows us to use Python’s standard “datetime” module while “django.utils import timezone” allows us to use Django’s time-zone-related utilities. Now open the Python shell again by running

```
py manage.py shell
```

and type in the following command lines:

```
>>> from polls.models import Choice, Question
# Make sure our __str__() addition worked.
>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by
keyword arguments.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith="What")
<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year=timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
```

```

<Question: What's up?>
# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
shortcut for primary-key exact lookups. The following is identical to
Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q=Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
Choice object, does the INSERT statement, adds the choice to the set of
available choices and returns the new Choice object. Django creates a set
(defined as "choice_set") to hold the "other side" of a ForeignKey
relation (e.g. a question's choice) which can be accessed via the API.
>>> q=Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text="Not much", votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text="The sky", votes=0)
<Choice: The sky>
>>> c=q.choice_set.create(choice_text="Just hacking", votes=0)
<Choice: Not much>

```

```

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking>]>
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need. Use
double underscores to separate relationships. This works as many levels
deep as you want; there's no limit. Find all Choices for any question
whose pub_date is in this year (reusing the 'current_year' variable we
created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking>]>

# Let's delete one of the choices.
>>> c=q.choice_set.filter(choice_text__startswith="Just")
>>>c.delete()
# The "Just hacking" choice has been deleted

>>>q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>]>

```

What we've done is essentially create several choices to choose from as well as learn how to filter between the different choices. Now, we'll create an admin user for our server. Run:

```
py manage.py createsuperuser
```

and then enter your username, email and password (the second time as confirmation). You can then start the server with the "runserver" command and visit <http://127.0.0.1:8000/admin/> (change the port number accordingly if necessary). Once there, you'll need to enter your username and password to reach the main page. Here you can see the "AUTHENTICATION AND AUTHORISATION" section that's provided by Django's authentication system. However, our "polls" app is seemingly missing; To remedy this, open polls/admin.py and edit the code as such:

```
from django.contrib import admin
```

```
from .models import Question
```

```
admin.site.register(Question)
```

Once your changes have been saved, reload your page and a “POLL” section should appear. You can then click on “Questions” and click on the “What’s up?” question and edit it however you like. The editing form you see here was generated from the Question model and its different field types (“DateTimeField” and “CharField”) each have their own unique HTML input widget. The “Today” and “Now” shortcuts that you see are actually JavaScript shortcuts.

If you chose to make any changes to your question and saved them, you can see exactly what changes you made by clicking the “History” button at the top right corner in the same page where you edit the question.