

[Django models](http://tutorial.djangogirls.org/en)

**Django models**

What we want to create now is something that will store all the posts in our blog. But to be able to do that we need to talk a little bit about things called objects.

**Objects**

There is a concept in programming called Object-oriented programming. The idea is that instead of writing everything as a boring sequence of programming instructions we can model things and define how they interact with each other.

So what is an object? It is a collection of properties and actions. It sounds weird, but we will give you an example.

If we want to model a cat we will create an object Cat that has some properties such as: color, age,mood (like good, bad, or sleepy ;)), and owner (that is a Person object or maybe, in case of a stray cat, this property is empty).

Then the Cat has some actions: purr, scratch, or feed (in which case, we will give the cat someCatFood, which could be a separate object with properties, like taste).

Cat

--------

color

age

mood

owner

purr()

scratch()

feed(cat\_food)

CatFood

--------

taste

So basically the idea is to describe real things in code with properties (called object properties) and actions (called methods).

How will we model blog posts then? We want to build a blog, right?

We need to answer the question: What is a blog post? What properties should it have?

Well, for sure our blog post needs some text with its content and a title, right? It would be also nice to know who wrote it - so we need an author. Finally, we want to know when the post was created and published.

Post

--------

title

text

author

created\_date

published\_date

What kind of things could be done with a blog post? It would be nice to have some method that publishes the post, right?

So we will need a publish method.

Since we already know what we want to achieve, let's start modeling it in Django!

**Django model**

Knowing what an object is, we can create a Django model for our blog post.

A model in Django is a special kind of object - it is saved in the database. A database is a collection of data. This is a place in which you will store information about users, your blog posts, etc. We will be using a SQLite database to store our data. This is the default Django database adapter -- it'll be enough for us right now.

You can think of a model in the database as a spreadsheet with columns (fields) and rows (data).

**Creating an application**

To keep everything tidy, we will create a separate application inside our project. It is very nice to have everything organized from the very beginning. To create an application we need to run the following command in the console (from djangogirls directory where manage.py file is):

(myvenv) ~/djangogirls$ python manage.py startapp blog

You will notice that a new blog directory is created and it contains a number of files now. Our directories and files in our project should look like this:

djangogirls

├── mysite

| \_\_init\_\_.py

| settings.py

| urls.py

| wsgi.py

├── manage.py

└── blog

├── migrations

| \_\_init\_\_.py

├── \_\_init\_\_.py

├── admin.py

├── models.py

├── tests.py

└── views.py

After creating an application we also need to tell Django that it should use it. We do that in the filemysite/settings.py. We need to find INSTALLED\_APPS and add a line containing 'blog', just above). So the final product 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',

)

**Creating a blog post model**

In the blog/models.py file we define all objects called Models - this is a place in which we will define our blog post.

Let's open blog/models.py, remove everything from it and write code like this:

from django.db import models

from django.utils import timezone

class Post(models.Model):

author = models.ForeignKey('auth.User')

title = models.CharField(max\_length=200)

text = models.TextField()

created\_date = models.DateTimeField(

default=timezone.now)

published\_date = models.DateTimeField(

blank=True, null=True)

def publish(self):

self.published\_date = timezone.now()

self.save()

def \_\_str\_\_(self):

return self.title

Double-check that you use two underscore characters (\_) on each side of str. This convention is used frequently in Python and sometimes we also call them "dunder" (short for "double-underscore").

It looks scary, right? But no worries we will explain what these lines mean!

All lines starting with from or import are lines that add some bits from other files. So instead of copying and pasting the same things in every file, we can include some parts with from ... import ....

class Post(models.Model): - this line defines our model (it is an object).

* class is a special keyword that indicates that we are defining an object.
* Post is the name of our model. We can give it a different name (but we must avoid special characters and whitespaces). Always start a class name with an uppercase letter.
* models.Model means that the Post is a Django Model, so Django knows that it should be saved in the database.

Now we define the properties we were talking about: title, text, created\_date, published\_dateand author. To do that we need to define a type of each field (Is it text? A number? A date? A relation to another object, like a User?).

* models.CharField - this is how you define text with a limited number of characters.
* models.TextField - this is for long text without a limit. Sounds ideal for blog post content, right?
* models.DateTimeField - this is a date and time.
* models.ForeignKey - this is a link to another model.

We will not explain every bit of code here since it would take too much time. You should take a look at Django's documentation if you want to know more about Model fields and how to define things other than those described above (<https://docs.djangoproject.com/en/1.9/ref/models/fields/#field-types>).

What about def publish(self):? It is exactly the publish method we were talking about before. defmeans that this is a function/method and publish is the name of the method. You can change the name of the method, if you want. The naming rule is that we use lowercase and underscores instead of whitespaces. For example, a method that calculates average price could be calledcalculate\_average\_price.

Methods often return something. There is an example of that in the \_\_str\_\_ method. In this scenario, when we call \_\_str\_\_() we will get a text (**string**) with a Post title.

Also notice that both def publish(self):, and def \_\_str\_\_(self): are indented inside our class. Because Python is sensitive to whitespace, we need to indent our methods inside the class. Otherwise, the methods won't belong to the class, and you can get some unexpected behavior.

If something is still not clear about models, feel free to ask your coach! We know it is complicated, especially when you learn what objects and functions are at the same time. But hopefully it looks slightly less magic for you now!

**Create tables for models in your database**

The last step here is to add our new model to our database. First we have to make Django know that we have some changes in our model (we have just created it!). Go to your console window and type python manage.py makemigrations blog. It will look like this:

(myvenv) ~/djangogirls$ python manage.py makemigrations blog

Migrations for 'blog':

0001\_initial.py:

- Create model Post

Django prepared for us a migration file that we have to apply now to our database. Type python manage.py migrate blog and the output should be:

(myvenv) ~/djangogirls$ python manage.py migrate blog

Operations to perform:

Apply all migrations: blog

Running migrations:

Rendering model states... DONE

Applying blog.0001\_initial... OK

Hurray! Our Post model is now in our database! It would be nice to see it, right? Jump to the next chapter to see what your Post looks like!



[Your first Django project!](http://tutorial.djangogirls.org/en)

**Your first Django project!**

Part of this chapter is based on tutorials by Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parts of this chapter are based on the [django-marcador tutorial](http://django-marcador.keimlink.de/" \t "_blank) licensed under Creative Commons Attribution-ShareAlike 4.0 International License. The django-marcador tutorial is copyrighted by Markus Zapke-Gründemann et al.

We're going to create a simple blog!

The first step is to start a new Django project. Basically, this means that we'll run some scripts provided by Django that will create the skeleton of a Django project for us. This is just a bunch of directories and files that we will use later.

The names of some files and directories are very important for Django. You should not rename the files that we are about to create. Moving them to a different place is also not a good idea. Django needs to maintain a certain structure to be able to find important things.

Remember to run everything in the virtualenv. If you don't see a prefix (myvenv) in your console you need to activate your virtualenv. We explained how to do that in the **Django installation** chapter in the **Working with virtualenv** part. Typing myvenv\Scripts\activate on Windows or source myvenv/bin/activate on Mac OS / Linux will do this for you.

In your MacOS or Linux console you should run the following command; **don't forget to add the period (or dot) . at the end**:

(myvenv) ~/djangogirls$ django-admin startproject mysite .

On Windows; **don't forget to add the period (or dot) . at the end**:

(myvenv) C:\Users\Name\djangogirls> django-admin startproject mysite .

The period . is crucial because it tells the script to install Django in your current directory (for which the period . is a short-hand reference)

**Note** When typing the commands above, remember that you only type the part which starts django-admin or django-admin.py. The(myvenv) ~/djangogirls$ and (myvenv) C:\Users\Name\djangogirls> parts shown here are just examples of the prompt that will be inviting your input on your command line.

django-admin.py is a script that will create the directories and files for you. You should now have a directory structure which looks like this:

djangogirls

├───manage.py

└───mysite

settings.py

urls.py

wsgi.py

\_\_init\_\_.py

manage.py is a script that helps with management of the site. With it we will be able to start a web server on our computer without installing anything else, amongst other things.

The settings.py file contains the configuration of your website.

Remember when we talked about a mail carrier checking where to deliver a letter? urls.py file contains a list of patterns used by urlresolver.

Let's ignore the other files for now as we won't change them. The only thing to remember is not to delete them by accident!

**Changing settings**

Let's make some changes in mysite/settings.py. Open the file using the code editor you installed earlier.

It would be nice to have the correct time on our website. Go to the [wikipedia timezones list](http://en.wikipedia.org/wiki/List_of_tz_database_time_zones" \t "_blank) and copy your relevant time zone (TZ). (eg. Europe/Berlin )

In settings.py, find the line that contains TIME\_ZONE and modify it to choose your own timezone:

TIME\_ZONE = 'Europe/Berlin'

Modifying "Europe/Berlin" as appropriate

We'll also need to add a path for static files (we'll find out all about static files and CSS later in the tutorial). Go down to the *end* of the file, and just underneath the STATIC\_URL entry, add a new one calledSTATIC\_ROOT:

STATIC\_URL = '/static/'

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

**Setup a database**

There's a lot of different database software that can store data for your site. We'll use the default one,sqlite3.

This is already set up in this part of your mysite/settings.py file:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

To create a database for our blog, let's run the following in the console: python manage.py migrate (we need to be in the djangogirls directory that contains the manage.py file). If that goes well, you should see something like this:

(myvenv) ~/djangogirls$ python manage.py migrate

Operations to perform:

Apply all migrations: auth, admin, contenttypes, sessions

Running migrations:

Rendering model states... DONE

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

Applying contenttypes.0002\_remove\_content\_type\_name... OK

Applying auth.0002\_alter\_permission\_name\_max\_length... OK

Applying auth.0003\_alter\_user\_email\_max\_length... OK

Applying auth.0004\_alter\_user\_username\_opts... OK

Applying auth.0005\_alter\_user\_last\_login\_null... OK

Applying auth.0006\_require\_contenttypes\_0002... OK

Applying auth.0007\_alter\_validators\_add\_error\_messages... OK

Applying sessions.0001\_initial... OK

And we're done! Time to start the web server and see if our website is working!

You need to be in the directory that contains the manage.py file (the djangogirls directory). In the console, we can start the web server by running python manage.py runserver:

(myvenv) ~/djangogirls$ python manage.py runserver

If you are on Windows and this fails with UnicodeDecodeError, use this command instead:

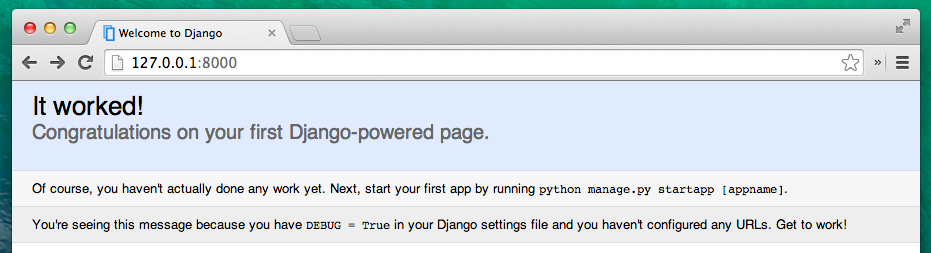
(myvenv) ~/djangogirls$ python manage.py runserver 0:8000

Now all you need to do is check that your website is running. Open your browser (Firefox, Chrome, Safari, Internet Explorer or whatever you use) and enter the address:

http://127.0.0.1:8000/

The web server will take over your command prompt until you stop it. To type more commands whilst it is running open a new terminal window and activate your virtualenv. To stop the web server, switch back to the window in which it's running and pressing CTRL+C - Control and C buttons together (on Windows, you might have to press Ctrl+Break).

Congratulations! You've just created your first website and run it using a web server! Isn't that awesome?



Ready for the next step? It's time to create some content!

# Deploy!

**Note** The following chapter can be sometimes a bit hard to get through. Persist and finish it; deployment is an important part of the website development process. This chapter is placed in the middle of the tutorial so that your mentor can help with the slightly trickier process of getting your website online. This means you can still finish the tutorial on your own if you run out of time.

Until now, your website was only available on your computer. Now you will learn how to deploy it! Deploying is the process of publishing your application on the Internet so people can finally go and see your app :).

As you learned, a website has to be located on a server. There are a lot of server providers available on the internet. We will use one that has a relatively simple deployment process: [PythonAnywhere](http://pythonanywhere.com/). PythonAnywhere is free for small applications that don't have too many visitors so it'll definitely be enough for you now.

The other external service we'll be using is [GitHub](http://www.github.com/), which is a code hosting service. There are others out there, but almost all programmers have a GitHub account these days, and now so will you!

These three places will be important to you. Your local computer will be the place where you do development and testing. When you're happy with the changes, you will place a copy of your program on GitHub. Your website will be on PythonAnywhere and you will update it by getting a new copy of your code from GitHub.

# Git

Git is a "version control system" used by a lot of programmers. This software can track changes to files over time so that you can recall specific versions later. A bit like the "track changes" feature in Microsoft Word, but much more powerful.

## Installing Git

**Note** If you already did the Installation steps, no need to do this again - you can skip to the next section and start creating your Git repository.

### Windows

You can download Git from [git-scm.com](http://git-scm.com/). You can hit "next next next" on all steps except for one; in the 5th step entitled "Adjusting your PATH environment", choose "Run Git and associated Unix tools from the Windows command-line" (the bottom option). Other than that, the defaults are fine. Checkout Windows-style, commit Unix-style line endings is good.

### MacOS

Download Git from [git-scm.com](http://git-scm.com/) and just follow the instructions.

### Linux

If it isn't installed already, git should be available via your package manager, so try:

#### Debian or Ubuntu

$ sudo apt-get install git

#### Fedora (up to 21)

$ sudo yum install git

#### Fedora (22+)

$ sudo dnf install git

## Starting our Git repository

Git tracks changes to a particular set of files in what's called a code repository (or "repo" for short). Let's start one for our project. Open up your console and run these commands, in the djangogirls directory:

**Note** Check your current working directory with a pwd (OSX/Linux) or cd (Windows) command before initializing the repository. You should be in the djangogirls folder.

$ git init

Initialized empty Git repository in ~/djangogirls/.git/

$ git config --global user.name "Your Name"

$ git config --global user.email you@example.com

Initializing the git repository is something we only need to do once per project (and you won't have to re-enter the username and email again ever).

Git will track changes to all the files and folders in this directory, but there are some files we want it to ignore. We do this by creating a file called .gitignore in the base directory. Open up your editor and create a new file with the following contents:

\*.pyc

\_\_pycache\_\_

myvenv

db.sqlite3

/static

.DS\_Store

And save it as .gitignore in the "djangogirls" folder.

**Note** The dot at the beginning of the file name is important! If you're having any difficulty creating it (Macs don't like you to create files that begin with a dot via the Finder, for example), then use the "Save As" feature in your editor, it's bulletproof.

**Note** One of the files you specified in your.gitignore file is db.sqlite3. That file is your local database, where all or your posts are stored. We don't want to add this to your repository, because your website on PythonAnywhere is going to be using a different database. That database could be SQLite, like your development machine, but usually, you will use one called MySQL which can deal with a lot more site visitors than SQLite. Either way, by ignoring your SQLite database for the GitHub copy, it means that all of the posts you created so far are going to stay and only be available locally, but you're gonna have to add them again on production. You should think of your local database as a good playground where you can test different things and not be afraid that you're going to delete your real posts from your blog.

It's a good idea to use a git status command before git add or whenever you find yourself unsure of what has changed. This will help stop any surprises from happening, such as wrong files being added or commited. The git status command returns information about any untracked/modifed/staged files, branch status, and much more. The output should be similar to:

$ git status

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

.gitignore

blog/

manage.py

mysite/

nothing added to commit but untracked files present (use "git add" to track)

And finally we save our changes. Go to your console and run these commands:

$ git add -A .

$ git commit -m "My Django Girls app, first commit"

[...]

13 files changed, 200 insertions(+)

create mode 100644 .gitignore

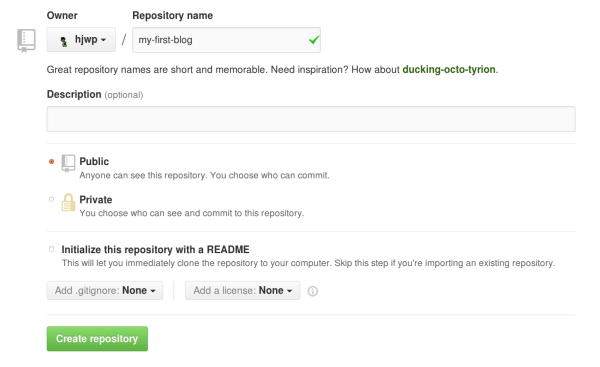
[...]

create mode 100644 mysite/wsgi.py

## Pushing our code to GitHub

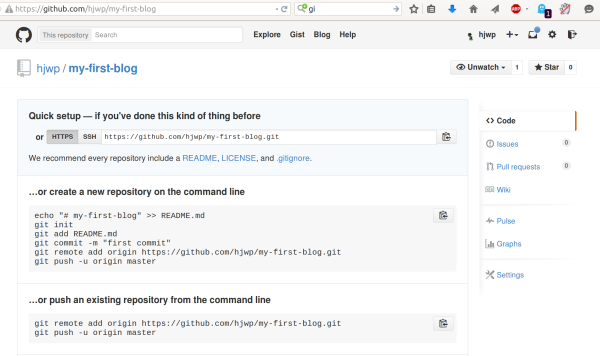
Go to [GitHub.com](http://www.github.com/) and sign up for a new, free user account. (If you already did that in the workshop prep, that is great!)

Then, create a new repository, giving it the name "my-first-blog". Leave the "initialise with a README" tickbox un-checked, leave the .gitignore option blank (we've done that manually) and leave the License as None.



**Note** The name my-first-blog is important -- you could choose something else, but it's going to occur lots of times in the instructions below, and you'd have to substitute it each time. It's probably easier to just stick with the name my-first-blog.

On the next screen, you'll be shown your repo's clone URL. Choose the "HTTPS" version, copy it, and we'll paste it into the terminal shortly:



Now we need to hook up the Git repository on your computer to the one up on GitHub.

Type the following into your console (Replace <your-github-username> with the username you entered when you created your GitHub account, but without the angle-brackets):

$ git remote add origin https://github.com/<your-github-username>/my-first-blog.git

$ git push -u origin master

Enter your GitHub username and password and you should see something like this:

Username for 'https://github.com': hjwp

Password for 'https://hjwp@github.com':

Counting objects: 6, done.

Writing objects: 100% (6/6), 200 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/hjwp/my-first-blog.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

Your code is now on GitHub. Go and check it out! You'll find it's in fine company - [Django](https://github.com/django/django), the [Django Girls Tutorial](https://github.com/DjangoGirls/tutorial), and many other great open source software projects also host their code on GitHub :)

# Setting up our blog on PythonAnywhere

**Note** You might have already created a PythonAnywhere account earlier during the install steps - if so, no need to do it again.

Next it's time to sign up for a free "Beginner" account on PythonAnywhere.

* [www.pythonanywhere.com](https://www.pythonanywhere.com/)

**Note** When choosing your username here, bear in mind that your blog's URL will take the formyourusername.pythonanywhere.com, so either choose your own nickname, or a name for what your blog is all about.

## Pulling our code down on PythonAnywhere

When you've signed up for PythonAnywhere, you'll be taken to your dashboard or "Consoles" page. Choose the option to start a "Bash" console -- that's the PythonAnywhere version of a console, just like the one on your computer.

**Note** PythonAnywhere is based on Linux, so if you're on Windows, the console will look a little different from the one on your computer.

Let's pull down our code from GitHub and onto PythonAnywhere by creating a "clone" of our repo. Type the following into the console on PythonAnywhere (don't forget to use your GitHub username in place of<your-github-username>):

$ git clone https://github.com/<your-github-username>/my-first-blog.git

This will pull down a copy of your code onto PythonAnywhere. Check it out by typing tree my-first-blog:

$ tree my-first-blog

my-first-blog/

├── blog

│ ├── \_\_init\_\_.py

│ ├── admin.py

│ ├── migrations

│ │ ├── 0001\_initial.py

│ │ └── \_\_init\_\_.py

│ ├── models.py

│ ├── tests.py

│ └── views.py

├── manage.py

└── mysite

├── \_\_init\_\_.py

├── settings.py

├── urls.py

└── wsgi.py

### Creating a virtualenv on PythonAnywhere

Just like you did on your own computer, you can create a virtualenv on PythonAnywhere. In the Bash console, type:

$ cd my-first-blog

$ virtualenv --python=python3.4 myvenv

Running virtualenv with interpreter /usr/bin/python3.4

[...]

Installing setuptools, pip...done.

$ source myvenv/bin/activate

(myvenv) $ pip install django~=1.9.0

Collecting django

[...]

Successfully installed django-1.9

**Note** The pip install step can take a couple of minutes. Patience, patience! But if it takes more than 5 minutes, something is wrong. Ask your coach.

### Creating the database on PythonAnywhere

Here's another thing that's different between your own computer and the server: it uses a different database. So the user accounts and posts can be different on the server and on your computer.

We can initialise the database on the server just like we did the one on your own computer, with migrateand createsuperuser:

(myvenv) $ python manage.py migrate

Operations to perform:

[...]

Applying sessions.0001\_initial... OK

(myvenv) $ python manage.py createsuperuser

## Publishing our blog as a web app

Now our code is on PythonAnywhere, our virtualenv is ready, and the database is initialised. We're ready to publish it as a web app!

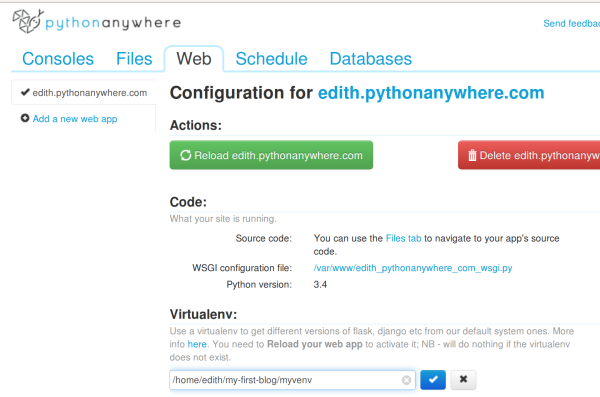
Click back to the PythonAnywhere dashboard by clicking on its logo, and go click on the **Web** tab. Finally, hit **Add a new web app**.

After confirming your domain name, choose **manual configuration** (NB not the "Django" option) in the dialog. Next choose **Python 3.4**, and click Next to finish the wizard.

**Note** Make sure you choose the "Manual configuration" option, not the "Django" one. We're too cool for the default PythonAnywhere Django setup ;-)

### Setting the virtualenv

You'll be taken to the PythonAnywhere config screen for your webapp, which is where you'll need to go whenever you want to make changes to the app on the server.



In the "Virtualenv" section, click the red text that says "Enter the path to a virtualenv", and enter:/home/<your-username>/my-first-blog/myvenv/. Click the blue box with the check mark to save the path before moving on.

**Note** Substitute your own username as appropriate. If you make a mistake, PythonAnywhere will show you a little warning.

### Configuring the WSGI file

Django works using the "WSGI protocol", a standard for serving websites using Python, which PythonAnywhere supports. The way we configure PythonAnywhere to recognise our Django blog is by editing a WSGI configuration file.

Click on the "WSGI configuration file" link (in the "Code" section near the top of the page -- it'll be named something like /var/www/<your-username>\_pythonanywhere\_com\_wsgi.py), and you'll be taken to an editor.

Delete all the contents and replace them with something like this:

import os

import sys

path = '/home/<your-username>/my-first-blog' # use your own username here

if path not in sys.path:

sys.path.append(path)

os.environ['DJANGO\_SETTINGS\_MODULE'] = 'mysite.settings'

from django.core.wsgi import get\_wsgi\_application

from django.contrib.staticfiles.handlers import StaticFilesHandler

application = StaticFilesHandler(get\_wsgi\_application())

**Note** Don't forget to substitute in your own username where it says <your-username> **Note** In line three, we make sure Python anywhere knows how to find our application. It is very important that this path name is correct, and especially that there are no extra spaces here. Otherwise you will see an "ImportError" in the error log.

This file's job is to tell PythonAnywhere where our web app lives and what the Django settings file's name is.

The StaticFilesHandler is for dealing with our CSS. This is taken care of automatically for you during local development by the runserver command. We'll find out a bit more about static files later in the tutorial, when we edit the CSS for our site.

Hit **Save** and then go back to the **Web** tab.

We're all done! Hit the big green **Reload** button and you'll be able to go view your application. You'll find a link to it at the top of the page.

## Debugging tips

If you see an error when you try to visit your site, the first place to look for some debugging info is in your**error log**. You'll find a link to this on the PythonAnywhere [Web tab](https://www.pythonanywhere.com/web_app_setup/). See if there are any error messages in there; the most recent ones are at the bottom. Common problems include:

* Forgetting one of the steps we did in the console: creating the virtualenv, activating it, installing Django into it, migrating the database.
* Making a mistake in the virtualenv path on the Web tab -- there will usually be a little red error message on there, if there is a problem.
* Making a mistake in the WSGI configuration file -- did you get the path to your my-first-blog folder right?
* Did you pick the same version of Python for your virtualenv as you did for your web app? Both should be 3.4.
* There are some [general debugging tips on the PythonAnywhere wiki](https://www.pythonanywhere.com/wiki/DebuggingImportError).

And remember, your coach is here to help!

# You are live!

The default page for your site should say "Welcome to Django", just like it does on your local computer. Try adding /admin/ to the end of the URL, and you'll be taken to the admin site. Log in with the username and password, and you'll see you can add new Posts on the server.

Once you have a few posts created, you can go back to your local setup (not PythonAnywhere). From here you should work on your local setup to make changes. This is a common workflow in Web development (make changes locally, push those changes to GitHub, pull your changes down to your live Web server). This allows you to work and experiment without breaking your live Web site. Pretty cool, huh?

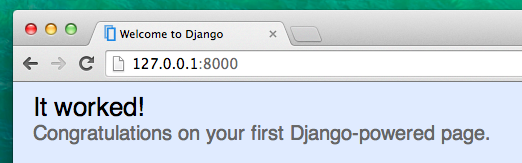
Give yourself a HUGE pat on the back! Server deployments are one of the trickiest parts of web development and it often takes people several days before they get them working. But you've got your site live, on the real Internet, just like that!

**Django urls**

We're about to build our first webpage: a homepage for your blog! But first, let's learn a little bit about Django urls.

**What is a URL?**

A URL is simply a web address. You can see a URL every time you visit a website - it is visible in your browser's address bar (yes! 127.0.0.1:8000 is a URL! And https://djangogirls.com is also a URL):



Every page on the Internet needs its own URL. This way your application knows what it should show to a user who opens a URL. In Django we use something called URLconf (URL configuration). URLconf is a set of patterns that Django will try to match with the received URL to find the correct view.

**How do URLs work in Django?**

Let's open up the mysite/urls.py file in your code editor of choice and see what it looks like:

"""mysite URL Configuration

[...]

"""

from django.conf.urls import url

from django.contrib import admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

]

As you can see, Django already put something here for us.

Lines between triple quotes (''' or """) are called docstrings - you can write them at the top of a file, class or method to describe what it does. They won't be run by Python.

The admin URL, which you visited in previous chapter is already here:

url(r'^admin/', admin.site.urls),

It means that for every URL that starts with admin/ Django will find a corresponding *view*. In this case we're including a lot of admin URLs so it isn't all packed into this small file -- it's more readable and cleaner.

**Regex**

Do you wonder how Django matches URLs to views? Well, this part is tricky. Django uses regex, short for "regular expressions". Regex has a lot (a lot!) of rules that form a search pattern. Since regexes are an advanced topic, we will not go in detail over how they work.

If you still wish to understand how we created the patterns, here is an example of the process - we will only need a limited subset of the rules to express the pattern we are looking for, namely:

^ for beginning of the text

$ for end of text

\d for a digit

+ to indicate that the previous item should be repeated at least once

() to capture part of the pattern

Anything else in the url definition will be taken literally.

Now imagine you have a website with the address like that: http://www.mysite.com/post/12345/, where12345 is the number of your post.

Writing separate views for all the post numbers would be really annoying. With regular expression we can create a pattern that will match the url and extract the number for us: ^post/(\d+)/$. Let's break it down piece by piece to see what we are doing here:

* **^post/** is telling Django to take anything that has post/ at the beginning of the url (right after ^)
* **(\d+)** means that there will be a number (one or more digits) and that we want the number captured and extracted
* **/** tells django that another / character should follow
* **$** then indicates the end of the URL meaning that only strings ending with the / will match this pattern

**Your first Django url!**

Time to create our first URL! We want '<http://127.0.0.1:8000/>' to be a homepage of our blog and display a list of posts.

We also want to keep the mysite/urls.py file clean, so we will import urls from our blog application to the main mysite/urls.py file.

Go ahead, add a line that will import blog.urls into the main url (''). Note that we are using theinclude function here so you will need to add that to the import on the first line of the file.

Your mysite/urls.py file should now look like this:

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'', include('blog.urls')),

]

Django will now redirect everything that comes into '<http://127.0.0.1:8000/>' to blog.urls and look for further instructions there.

When writing regular expressions in Python it is always done with r in front of the string. This is a helpful hint for Python that the string may contain special characters that are not meant for Python itself, but for the regular expression instead.

**blog.urls**

Create a new blog/urls.py empty file. All right! Add these two first lines:

from django.conf.urls import url

from . import views

Here we're importing Django's function url and all of our views from blog application (we don't have any yet, but we will get to that in a minute!)

After that, we can add our first URL pattern:

urlpatterns = [

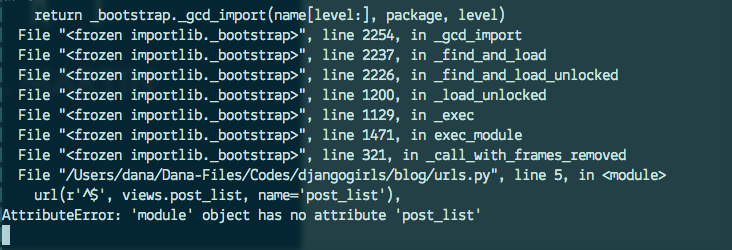
url(r'^$', views.post\_list, name='post\_list'),

]

As you can see, we're now assigning a view called post\_list to ^$ URL. This regular expression will match ^ (a beginning) followed by $ (an end) - so only an empty string will match. That's correct, because in Django URL resolvers, '<http://127.0.0.1:8000/>' is not a part of the URL. This pattern will tell Django that views.post\_list is the right place to go if someone enters your website at the '<http://127.0.0.1:8000/>' address.

The last part name='post\_list' is the name of the URL that will be used to identify the view. This can be the same as the name of the view but it can also be something completely different. We will be using the named URLs later in the project so it is important to name each URL in the app. We should also try to keep the names of URLs unique and easy to remember.

If you try to visit <http://127.0.0.1:8000/> now, then you'll find some sort of 'web page not available' message. This is because the server (remember typing runserver?) is no longer running. Take a look at your server console window to find out why.



Your console is showing an error but don't worry—they're actually pretty useful:

It's telling you that there is **no attribute 'post\_list'**. That's the name of the *view* that Django is trying to find and use, but we haven't created it yet. No worries, we will get there.

If you want to know more about Django URLconfs, look at the official documentation:[https://docs.djangoproject.com/en/1.9/topics/http/urls/](https://docs.djangoproject.com/en/1.9/topics/http/urls/" \t "_blank)

# Dynamic data in templates

We have different pieces in place: the Post model is defined in models.py, we have post\_list inviews.py and the template added. But how will we actually make our posts appear in our HTML template? Because that is what we want to do. Take some content (models saved in the database) and display it nicely in our template, right?

This is exactly what views are supposed to do: connect models and templates. In our post\_list view we will need to take models we want to display and pass them to the template. In a view we decide what (model) will be displayed in a template.

OK, so how will we achieve it?

We need to open our blog/views.py. So far post\_list view looks like this:

from django.shortcuts import render

def post\_list(request):

return render(request, 'blog/post\_list.html', {})

Remember when we talked about including code written in different files? Now it is the moment when we have to include the model we have written in models.py. We will add this line from .models import Postlike this:

from django.shortcuts import render

from .models import Post

The dot before models means current directory or current application. Both views.py and models.pyare in the same directory. This means we can use . and the name of the file (without .py). Then we import the name of the model (Post).

But what's next? To take actual blog posts from Post model we need something called QuerySet.

## QuerySet

You should already be familiar with how QuerySets work. We talked about it in [Django ORM (QuerySets) chapter](http://tutorial.djangogirls.org/en/django_orm/).

So now we want published blog posts sorted by published\_date, right? We already did that in QuerySets chapter!

Post.objects.filter(published\_date\_\_lte=timezone.now()).order\_by('published\_date')

Now we put this piece of code inside the blog/views.py file by adding it to the function def post\_list(request):

from django.shortcuts import render

from django.utils import timezone

from .models import Post

def post\_list(request):

posts = Post.objects.filter(published\_date\_\_lte=timezone.now()).order\_by('published\_date')

return render(request, 'blog/post\_list.html', {})

Please note that we create a variable for our QuerySet: posts. Treat this as the name of our QuerySet. From now on we can refer to it by this name.

Also, the code uses the timezone.now() function, so we need to add an import for timezone.

The last missing part is passing the posts QuerySet to the template. Don't worry we will cover how to display it in a next chapter.

In the render function we already have parameter with request (so everything we receive from the user via the Internet) and a template file 'blog/post\_list.html'. The last parameter, which looks like this: {} is a place in which we can add some things for the template to use. We need to give them names (we will stick to 'posts' right now :)). It should look like this: {'posts': posts}. Please note that the part before : is a string; you need to wrap it with quotes ''.

So finally our blog/views.py file should look like this:

from django.shortcuts import render

from django.utils import timezone

from .models import Post

def post\_list(request):

posts = Post.objects.filter(published\_date\_\_lte=timezone.now()).order\_by('published\_date')

return render(request, 'blog/post\_list.html', {'posts': posts})

That's it! Time to go back to our template and display this QuerySet!

Want to read a little bit more about QuerySets in Django? You should look here:[https://docs.djangoproject.com/en/1.9/ref/models/querysets/](https://docs.djangoproject.com/en/1.9/ref/models/querysets/" \t "_blank)

**Django templates**

Time to display some data! Django gives us some helpful built-in **template tags** for that.

**What are template tags?**

You see, in HTML, you can't really write Python code, because browsers don't understand it. They only know HTML. We know that HTML is rather static, while Python is much more dynamic.

**Django template tags** allow us to transfer Python-like things into HTML, so you can build dynamic websites faster and easier. Yikes!

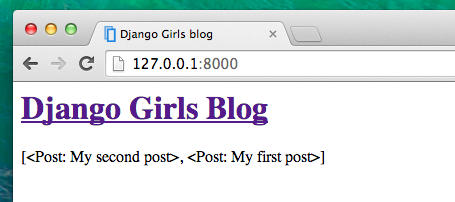
**Display post list template**

In the previous chapter we gave our template a list of posts in the posts variable. Now we will display it in HTML.

To print a variable in Django templates, we use double curly brackets with the variable's name inside, like this:

{{ posts }}

Try this in your blog/templates/blog/post\_list.html template. Replace everything from the second<div> to the third </div> with {{ posts }}. Save the file, and refresh the page to see the results:



As you can see, all we've got is this:

[<Post: My second post>, <Post: My first post>]

This means that Django understands it as a list of objects. Remember from **Introduction to Python** how we can display lists? Yes, with for loops! In a Django template you do them like this:

{% for post in posts %}

{{ post }}

{% endfor %}

Try this in your template.



It works! But we want them to be displayed like the static posts we created earlier in the **Introduction to HTML** chapter. You can mix HTML and template tags. Our body will look like this:

<div>

<h1><a href="/">Django Girls Blog</a></h1>

</div>

{% for post in posts %}

<div>

<p>published: {{ post.published\_date }}</p>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endfor %}

Everything you put between {% for %} and {% endfor %} will be repeated for each object in the list. Refresh your page:



Have you noticed that we used a slightly different notation this time {{ post.title }} or {{ post.text }}? We are accessing data in each of the fields defined in our Post model. Also the |linebreaks is piping the posts' text through a filter to convert line-breaks into paragraphs.

**One more thing**

It'd be good to see if your website will still be working on the public Internet, right? Let's try deploying to PythonAnywhere again. Here's a recap of the steps...

* First, push your code to Github

$ git status

[...]

$ git add -A .

$ git status

[...]

$ git commit -m "Modified templates to display posts from database."

[...]

$ git push

* Then, log back in to [PythonAnywhere](https://www.pythonanywhere.com/consoles/) and go to your **Bash console** (or start a new one), and run:

$ cd my-first-blog

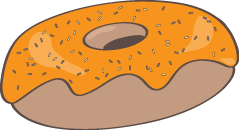
$ git pull

[...]

* Finally, hop on over to the [Web tab](https://www.pythonanywhere.com/web_app_setup/) and hit **Reload** on your web app. Your update should be live! If the blog posts on your PythonAnywhere site don't match the posts appearing on the blog hosted on your local server - that's OK. The databases on your local computer and Python Anywhere don't sync with the rest of your files.

Congrats! Now go ahead and try adding a new post in your Django admin (remember to add published\_date!) Make sure you are in the Django admin for your pythonanywhere site,[https://www.yourname.pythonanywhere.com/admin](https://www.yourname.pythonanywhere.com/admin" \t "_blank). Then refresh your page to see if the post appears there.

Works like a charm? We're proud! Step away from your computer for a bit, you have earned a break. :)



# Template extending

Another nice thing Django has for you is **template extending**. What does this mean? It means that you can use the same parts of your HTML for different pages of your website.

Templates help when you want to use the same information/layout in more than one place. You don't have to repeat yourself in every file. And if you want to change something, you don't have to do it in every template, just once!

## Create base template

A base template is the most basic template that you extend on every page of your website.

Let's create a base.html file in blog/templates/blog/:

blog

└───templates

└───blog

base.html

post\_list.html

Then open it up and copy everything from post\_list.html to base.html file, like this:

{% load staticfiles %}

<html>

<head>

<title>Django Girls blog</title>

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

<link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

</head>

<body>

<div class="page-header">

<h1><a href="/">Django Girls Blog</a></h1>

</div>

<div class="content container">

<div class="row">

<div class="col-md-8">

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endfor %}

</div>

</div>

</div>

</body>

</html>

Then in base.html, replace your whole <body> (everything between <body> and </body>) with this:

<body>

<div class="page-header">

<h1><a href="/">Django Girls Blog</a></h1>

</div>

<div class="content container">

<div class="row">

<div class="col-md-8">

{% block content %}

{% endblock %}

</div>

</div>

</div>

</body>

You might notice this replaced everything from {% for post in posts %} to {% endfor %} with:

{% block content %}

{% endblock %}

But why? You just created a block! You used the template tag {% block %} to make an area that will have HTML inserted in it. That HTML will come from another templates that extends this template (base.html). We will show you how to do this in a moment.

Now save base.html, and open your blog/templates/blog/post\_list.html again. You're going to remove everything above {% for post in posts %} and below {% endfor %}. When you're done the file will look like this:

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endfor %}

We want to use this as part of our template for all the content blocks. Time to add block tags to this file!

You want your block tag to match the tag in your base.html file. You also want it to include all the code that belongs in your content blocks. To do that, put everything between {% block content %} and {% endblock content %}. Like this:

{% block content %}

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endfor %}

{% endblock %}

Only one thing left. We need to connect these two templates together. This is what extending templates is all about! We'll do this by adding an extends tag to the beginning of the file. Like this:

{% extends 'blog/base.html' %}

{% block content %}

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endfor %}

{% endblock %}

That's it! Check if your website is still working properly :)

If you have an error TemplateDoesNotExist that says that there is no blog/base.html file and you have runserver running in the console, try to stop it (by pressing Ctrl+C - Control and C buttons together) and restart it by running a python manage.py runserver command.

**Extend your application**

We've already completed all the different steps necessary for the creation of our website: we know how to write a model, url, view and template. We also know how to make our website pretty.

Time to practice!

The first thing we need in our blog is, obviously, a page to display one post, right?

We already have a Post model, so we don't need to add anything to models.py.

**Create a template link to a post's detail**

We will start with adding a link inside blog/templates/blog/post\_list.html file. So far it should look like:

{% extends 'blog/base.html' %}

{% block content %}

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endfor %}

{% endblock content %}

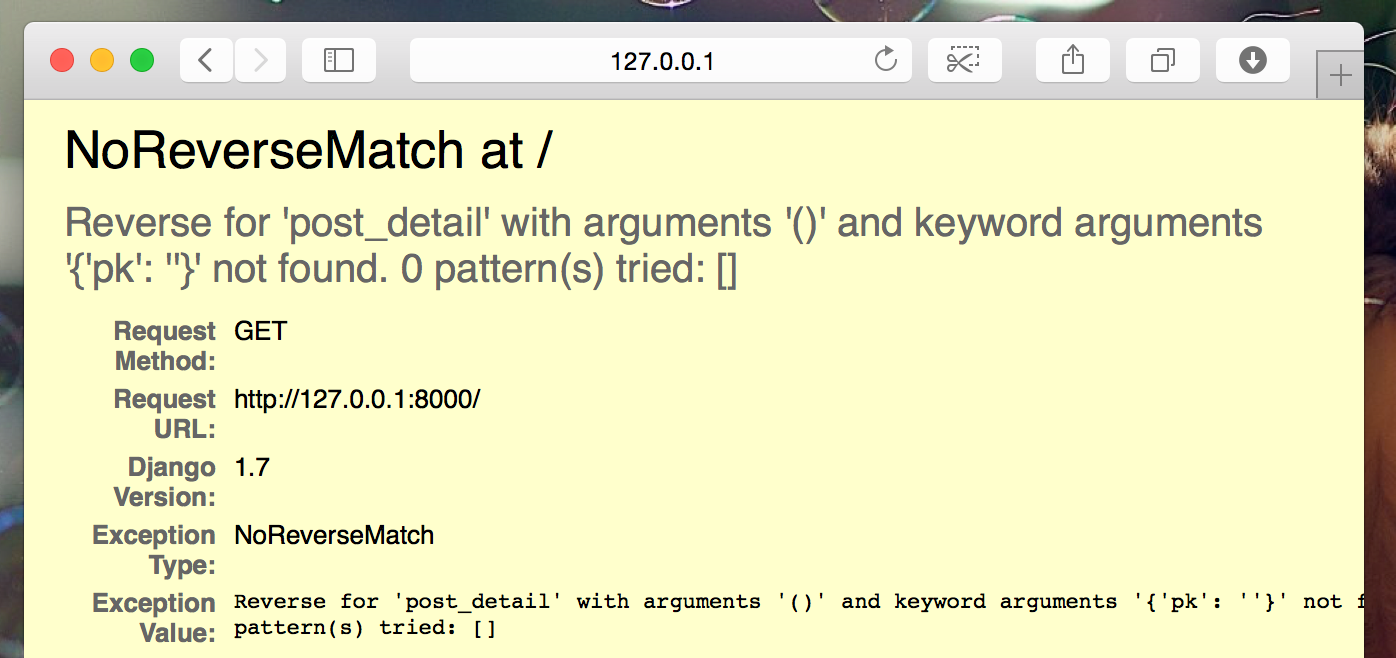
We want to have a link from a post's title in the post list to the post's detail page. Let's change <h1><a href="">{{ post.title }}</a></h1> so that it links to the post's detail page:

<h1><a href="{% url 'post\_detail' pk=post.pk %}">{{ post.title }}</a></h1>

Time to explain the mysterious {% url 'post\_detail' pk=post.pk %}. As you might suspect, the {% %}notation means that we are using Django template tags. This time we will use one that will create a URL for us!

blog.views.post\_detail is a path to a post\_detail *view* we want to create. Please note: blog is the name of our application (the directory blog), views is from the name of the views.py file and the last bit - post\_detail - is the name of the *view*.

Now when we go to: <http://127.0.0.1:8000/> we will have an error (as expected, since we don't have a URL or a *view* for post\_detail). It will look like this:



**Create a URL to a post's detail**

Let's create a URL in urls.py for our post\_detail *view*!

We want our first post's detail to be displayed at this **URL**: <http://127.0.0.1:8000/post/1/>

Let's make a URL in the blog/urls.py file to point Django to a *view* named post\_detail, that will show an entire blog post. Add the line url(r'^post/(?P<pk>\d+)/$', views.post\_detail, name='post\_detail'),to the blog/urls.py file. The file should look like this:

from django.conf.urls import include, url

from . import views

urlpatterns = [

url(r'^$', views.post\_list, name='post\_list'),

url(r'^post/(?P<pk>\d+)/$', views.post\_detail, name='post\_detail'),

]

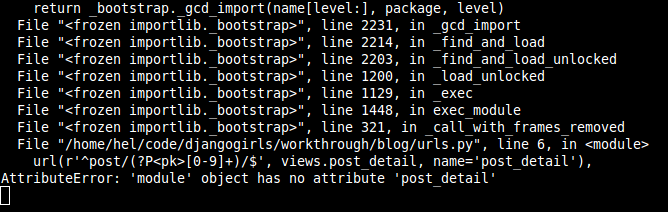
This part ^post/(?P<pk>\d+)/$ looks scary, but no worries - we will explain it for you:

* it starts with ^ again -- "the beginning"
* post/ only means that after the beginning, the URL should contain the word **post** and **/**. So far so good.
* (?P<pk>\d+) - this part is trickier. It means that Django will take everything that you place here and transfer it to a view as a variable called pk. \d also tells us that it can only be a digit, not a letter (so everything between 0 and 9). + means that there needs to be one or more digits there. So something like http://127.0.0.1:8000/post// is not valid, buthttp://127.0.0.1:8000/post/1234567890/ is perfectly ok!
* / - then we need **/** again
* $ - "the end"!

That means if you enter http://127.0.0.1:8000/post/5/ into your browser, Django will understand that you are looking for a *view* called post\_detail and transfer the information that pk equals 5 to that*view*.

pk is shortcut for primary key. This name is often used in Django projects. But you can name your variable as you like (remember: lowercase and \_ instead of whitespaces!). For example instead of (?P<pk>\d+) we could have variable post\_id, so this bit would look like: (?P<post\_id>\d+).

Ok, we've added a new URL pattern to blog/urls.py! Let's refresh the page: <http://127.0.0.1:8000/>Boom! The server has stopped running again. Have a look at the console - as expected, there's yet another error!



Do you remember what the next step is? Of course: adding a view!

**Add a post's detail view**

This time our *view* is given an extra parameter pk. Our *view* needs to catch it, right? So we will define our function as def post\_detail(request, pk):. Note that we need to use exactly the same name as the one we specified in urls (pk). Omitting this variable is incorrect and will result in an error!

Now, we want to get one and only one blog post. To do this we can use querysets like this:

Post.objects.get(pk=pk)

But this code has a problem. If there is no Post with given primary key (pk) we will have a super ugly error!



We don't want that! But, of course, Django comes with something that will handle that for us:get\_object\_or\_404. In case there is no Post with the given pk it will display much nicer page (calledPage Not Found 404 page).



The good news is that you can actually create your own Page not found page and make it as pretty as you want. But it's not super important right now, so we will skip it.

Ok, time to add a *view* to our views.py file!

We should open blog/views.py and add the following code:

from django.shortcuts import render, get\_object\_or\_404

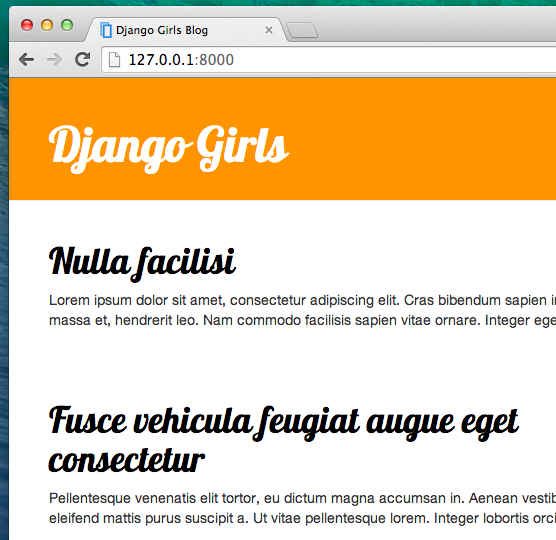
Near other from lines. And at the end of the file we will add our *view*:

def post\_detail(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

return render(request, 'blog/post\_detail.html', {'post': post})

Yes. It is time to refresh the page: <http://127.0.0.1:8000/>



It worked! But what happens when you click a link in blog post title?



Oh no! Another error! But we already know how to deal with it, right? We need to add a template!

**Create a template for post detail**

We will create a file in blog/templates/blog called post\_detail.html.

It will look like this:

{% extends 'blog/base.html' %}

{% block content %}

<div class="post">

{% if post.published\_date %}

<div class="date">

{{ post.published\_date }}

</div>

{% endif %}

<h1>{{ post.title }}</h1>

<p>{{ post.text|linebreaks }}</p>

</div>

{% endblock %}

Once again we are extending base.html. In the content block we want to display a post's published\_date (if it exists), title and text. But we should discuss some important things, right?

{% if ... %} ... {% endif %} is a template tag we can use when we want to check something (remember if ... else .. from **Introduction to Python** chapter?). In this scenario we want to check if a post's published\_date is not empty.

Ok, we can refresh our page and see if TemplateDoesNotExist is gone now.



Yay! It works!

**One more thing: deploy time!**

It'd be good to see if your website will still be working on PythonAnywhere, right? Let's try deploying again.

$ git status

$ git add -A .

$ git status

$ git commit -m "Added view and template for detailed blog post as well as CSS for the site."

$ git push

* Then, in a [PythonAnywhere Bash console](https://www.pythonanywhere.com/consoles/):

$ cd my-first-blog

$ git pull

[...]

* Finally, hop on over to the [Web tab](https://www.pythonanywhere.com/web_app_setup/) and hit **Reload**.

And that should be it! Congrats :)