**Install Python**

First check whether your computer is running a 32-bit version or a 64-bit version of Windows, on the "System type" line of the System Info page. To reach this page, try one of these methods:

* Press the Windows key and Pause/Break key at the same time
* Open your Control Panel from the Windows menu, then navigate to System & Security, then System
* Press the Windows button, then navigate to Settings > System > About

You can download Python for Windows from the website <https://www.python.org/downloads/windows/>. Click on the "Latest Python 3 Release - Python x.x.x" link. If your computer is running a **64-bit** version of Windows, download the **Windows x86-64 executable installer**. Otherwise, download the **Windows x86 executable installer**. After downloading the installer, you should run it (double-click on it) and follow the instructions there.

One thing to watch out for: During the installation, you will notice a window marked "Setup". Make sure you tick the "Add Python 3.6 to PATH" or 'Add Python to your environment variables" checkbox and click on "Install Now", as shown here (it may look a bit different if you are installing a different version)

**Installing packages with requirements**

A requirements file keeps a list of dependencies to be installed using pip install:

First create a requirements.txt file inside of the django folder, using the code editor that you installed earlier. You do this by opening a new file in the code editor and then saving it as requirements.txt in the django/ folder. Your directory will look like this:

django

└───requirements.txt

In your django/requirements.txt file you should add the following text:

django/requirements.txt

Django~=2.0.6

Now, run  following commandto install Django.

pip install -r requirements.txt

(or)

python -m pip install -r requirements.txt

**Create project: Windows**

On Windows you should run the following command. **(Don't forget to add the period (or dot) . at the end)**:

command-line

**C:\Users\Name\django> django-admin.exe startproject PythonWeb .**

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).

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:

django

├───manage.py

├───PythonWeb

│ settings.py

│ urls.py

│ wsgi.py

│ \_\_init\_\_.py

└───requirements.txt

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

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.

## Changing settings

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

We'll 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 called STATIC\_ROOT:

PythonWeb/settings.py

STATIC\_URL = '/static/'

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

When DEBUG is True and ALLOWED\_HOSTS is empty, the host is validated against ['localhost', '127.0.0.1', '[::1]']. This won't match our hostname on PythonAnywhere once we deploy our application so we will change the following setting:

mysite/settings.py

ALLOWED\_HOSTS = ['127.0.0.1']

## Set up 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:

mysite/settings.py

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:

python manage.py migrate

## Starting the web server

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

command-line

/djangogirls> python manage.py runserver

# Django model

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 django directory where manage.py file is):

Windows:

C:\Users\Name\djangogirls> python manage.py startapp blog

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

djangogirls

├── blog

│ ├── \_\_init\_\_.py

│ ├── admin.py

│ ├── apps.py

│ ├── migrations

│ │ └── \_\_init\_\_.py

│ ├── models.py

│ ├── tests.py

│ └── views.py

├── db.sqlite3

├── manage.py

├── mysite

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

└── requirements.txt

After creating an application, we also need to tell Django that it should use it. We do that in the file mysite/settings.py -- open it in your code editor. We need to find INSTALLED\_APPS and add a line containing 'blog', just above ]. So the final product should look like this:

mysite/settings.py

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 in the code editor, remove everything from it, and write code like this:

blog/models.py

from django.conf import settings

from django.db import models

from django.utils import timezone

class Post(models.Model):

author = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

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

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 whitespace). 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.
* 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.

### 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:

command-line

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

Migrations for 'blog':

blog/migrations/0001\_initial.py:

- Create model Post

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

command-line

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

Operations to perform:

Apply all migrations: blog

Running migrations:

Applying blog.0001\_initial... OK

# Django admin

To add, edit and delete the posts we've just modeled, we will use Django admin.

Let's open the blog/admin.py file in the code editor and replace its contents with this:

blog/admin.py

from django.contrib import admin

from .models import Post

admin.site.register(Post)

To make our model visible on the admin page, we need to register the model with admin.site.register(Post).

OK, time to look at our Post model. Remember to run python manage.py runserver in the console to run the web server. Go to your browser and type the address <http://127.0.0.1:8000/admin/>.

To log in, you need to create a superuser - a user account that has control over everything on the site. Go back to the command line, type python manage.py createsuperuser, and press enter.

Windows:

(myvenv) C:\Users\Name\djangogirls> python manage.py createsuperuser

The output should look like this (where the username and email should be your own ones):

Username: ssi

Email address: amsasi78@gmail.com

Password:

Password (again):

Superuser created successfully.

Return to your browser. Log in with the superuser's credentials you chose; you should see the Django admin dashboard.

Go to Posts and experiment a little bit with it. Add five or six blog posts. Don't worry about the content.

# 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 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.org is also a URL.)

## 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/urls.py

"""mysite URL Configuration

[...]

"""

from django.contrib import admin

from django.urls import path

urlpatterns = [

path('admin/', admin.site.urls),

]

The admin URL, which we visited already:

mysite/urls.py

path('admin/', admin.site.urls),

This line 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.

## Your first Django URL!

Time to create our first URL! We want '<http://127.0.0.1:8000/>' to be the home page of our blog and to 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. You will also need to change the from django.urls…line because we are using the include function here, so you will need to add that import to the line.

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

mysite/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('blog.urls')),

]

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

## blog.urls

Create a new empty file named urls.py in the blog directory, and open it in the code editor. All right! Add these first two lines:

blog/urls.py

from django.urls import path

from . import views

Here we're importing Django's function path and all of our views from the 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:

blog/urls.py

urlpatterns = [

path('', views.post\_list, name='post\_list'),

]

As you can see, we're now assigning a view called post\_list to the root URL. This URL pattern will match an empty string and the Django URL resolver will ignore the domain name (i.e., <http://127.0.0.1:8000/>) that prefixes the full url path. 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.

# Django views – time to create!

A view is a place where we put the "logic" of our application. It will request information from the modelyou created before and pass it to a template. We'll create a template in the next chapter. Views are just Python functions that are a little bit more complicated than the ones we wrote in the **Introduction to Python** chapter.

Views are placed in the views.py file. We will add our views to the blog/views.py file.

## blog/views.py

OK, let's open up this file in our code editor and see what's in there:

blog/views.py

from django.shortcuts import render

# Create your views here.

Not too much stuff here yet.

Remember that lines starting with # are comments – this means that those lines won't be run by Python.

Let's create a view as the comment suggests. Add the following minimal view below it:

blog/views.py

def post\_list(request):

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

As you can see, we created a function (def) called post\_list that takes request and will returnthe value it gets from calling another function render that will render (put together) our template blog/post\_list.html.

Save the file, go to <http://127.0.0.1:8000/> and see what we've got.

## Your first template!

Creating a template means creating a template file. Everything is a file, right? You have probably noticed this already.

Templates are saved in blog/templates/blog directory. So first create a directory called templatesinside your blog directory. Then create another directory called blog inside your templates directory:

blog

└───templates

└───blog

(You might wonder why we need two directories both called blog – as you will discover later, this is a useful naming convention that makes life easier when things start to get more complicated.)

And now create a post\_list.html file (just leave it blank for now) inside the blog/templates/blogdirectory.

See how your website looks now: <http://127.0.0.1:8000/>

If you still have an error TemplateDoesNotExist, try to restart your server. Go to the command line, stop the server by pressing Ctrl+C (Control and C keys together) and start it again by running a python manage.py runserver command.

Open the new file in the code editor, and add the following:

blog/templates/blog/post\_list.html

<html>

<head>

<title>Django blog</title>

</head>

<body>

<div>

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

</div>

<div>

<p>published: 14.06.2014, 12:14</p>

<h2><a href="">My first post</a></h2>

<p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.</p>

</div>

<div>

<p>published: 14.06.2014, 12:14</p>

<h2><a href="">My second post</a></h2>

<p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut f.</p>

</div>

</body>

</html>

Visit it to find out: <http://127.0.0.1:8000/>