# What is Django?

**Django is a free and open source web application framework, written in Python. A web framework is a set of components that helps you to develop websites faster and easier.**

**When you're building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc.**

**Luckily for you, other people long ago noticed that web developers face similar problems when building a new site, so they teamed up and created frameworks (Django being one of them) that give you ready-made components to use.**

**Frameworks exist to save you from having to reinvent the wheel and to help alleviate some of the overhead when you’re building a new site.**

## Why do you need a framework?

**To understand what Django is actually for, we need to take a closer look at the servers. The first thing is that the server needs to know that you want it to serve you a web page.**

**Imagine a mailbox (port) which is monitored for incoming letters (requests). This is done by a web server. The web server reads the letter and then sends a response with a webpage. But when you want to send something, you need to have some content. And Django is something that helps you create the content.**

## What happens when someone requests a website from your server?

**When a request comes to a web server, it's passed to Django which tries to figure out what is actually requested. It takes a web page address first and tries to figure out what to do. This part is done by Django's urlresolver (note that a website address is called a URL – Uniform Resource Locator – so the name urlresolver makes sense). It is not very smart – it takes a list of patterns and tries to match the URL. Django checks patterns from top to bottom and if something is matched, then Django passes the request to the associated function (which is called view).**

**Imagine a mail carrier with a letter. She is walking down the street and checks each house number against the one on the letter. If it matches, she puts the letter there. This is how the urlresolver works!**

**In the view function, all the interesting things are done: we can look at a database to look for some information. Maybe the user asked to change something in the data? Like a letter saying, "Please change the description of my job." The view can check if you are allowed to do that, then update the job description for you and send back a message: "Done!" Then the view generates a response and Django can send it to the user's web browser.**

**The description above is a little bit simplified, but you don't need to know all the technical things yet. Having a general idea is enough.**

**So instead of diving too much into details, we will start creating something with Django and we will learn all the important parts along the way!**

**These files are created when you run this command: django-admin startproject mysite**

* **The outer mysite/ root directory is just a container for your project. Its name doesn’t matter to Django; you can rename it to anything you like.**
* **manage.py: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about manage.py in [django-admin and manage.py](https://docs.djangoproject.com/en/2.1/ref/django-admin/).**
* **The inner mysite/ directory is the actual Python package for your project. Its name is the Python package name you’ll need to use to import anything inside it (e.g. mysite.urls).**
* **mysite/\_\_init\_\_.py: An empty file that tells Python that this directory should be considered a Python package. If you’re a Python beginner, read**[**more about packages**](https://docs.python.org/3/tutorial/modules.html#tut-packages)**in the official Python docs.**
* **mysite/settings.py: Settings/configuration for this Django project. [Django settings](https://docs.djangoproject.com/en/2.1/topics/settings/) will tell you all about how settings work.**
* **mysite/urls.py: The URL declarations for this Django project; a “table of contents” of your Django-powered site. You can read more about URLs in**[**URL dispatcher**](https://docs.djangoproject.com/en/2.1/topics/http/urls/)**.**
* **mysite/wsgi.py: An entry-point for WSGI-compatible web servers to serve your project. See**[**How to deploy with WSGI**](https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/)**for more details.**

## The development server[¶](https://docs.djangoproject.com/en/2.1/intro/tutorial01/#the-development-server)

**Let’s verify your Django project works. Change into the outer mysite directory, if you haven’t already, and run the following commands: python manage.py runserver**

**Now that the server’s running, visit**[**http://127.0.0.1:8000/**](http://127.0.0.1:8000/)**with your Web browser. You’ll see a “Congratulations!” page, with a rocket taking off. It worked!**

**Include()**

**The**[**include()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.include)**function allows referencing other URLconfs. Whenever Django encounters**[**include()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.include)**, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.**

**The idea behind**[**include()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.include)**is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (polls/urls.py), they can be placed under “/polls/”, or under “/fun\_polls/”, or under “/content/polls/”, or any other path root, and the app will still work.**

**When to use**[**include()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.include)

**You should always use include() when you include other URL patterns.admin.site.urls is the only exception to this.**

**The**[**path()**](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path)**function is passed four arguments, two required: route and view, and two optional: kwargs, and name. At this point, it’s worth reviewing what these arguments are for.**

### [path()](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path) argument: route[¶](https://docs.djangoproject.com/en/2.1/intro/tutorial01/#path-argument-route)

**route is a string that contains a URL pattern. When processing a request, Django starts at the first pattern in urlpatterns and makes its way down the list, comparing the requested URL against each pattern until it finds one that matches.**

**Patterns don’t search GET and POST parameters, or the domain name. For example, in a request to https://www.example.com/myapp/, the URLconf will look for myapp/. In a request to https://www.example.com/myapp/?page=3, the URLconf will also look for myapp/.**

### [path()](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path) argument: view[¶](https://docs.djangoproject.com/en/2.1/intro/tutorial01/#path-argument-view)

**When Django finds a matching pattern, it calls the specified view function with an [HttpRequest](https://docs.djangoproject.com/en/2.1/ref/request-response/" \l "django.http.HttpRequest" \o "django.http.HttpRequest) object as the first argument and any “captured” values from the route as keyword arguments. We’ll give an example of this in a bit.**

### [path()](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path) argument: kwargs[¶](https://docs.djangoproject.com/en/2.1/intro/tutorial01/" \l "path-argument-kwargs" \o "Permalink to this headline)

**Arbitrary keyword arguments can be passed in a dictionary to the target view. We aren’t going to use this feature of Django in the tutorial.**

### [path()](https://docs.djangoproject.com/en/2.1/ref/urls/#django.urls.path) argument: name[¶](https://docs.djangoproject.com/en/2.1/intro/tutorial01/#path-argument-name)

**Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This powerful feature allows you to make global changes to the URL patterns of your project while only touching a single file.**

**When you’re comfortable with the basic request and response flow, read**[**part 2 of this tutorial**](https://docs.djangoproject.com/en/2.1/intro/tutorial02/)**to start working with the database.**

## Creating models[¶](https://docs.djangoproject.com/en/2.1/intro/tutorial02/#creating-models)

**Now we’ll define your models – essentially, your database layout, with additional metadata.**

**Philosophy**

**A model is the single, definitive source of truth about your data. It contains the essential fields and behaviors of the data you’re storing. Django follows the**[**DRY Principle**](https://docs.djangoproject.com/en/2.1/misc/design-philosophies/#dry)**. The goal is to define your data model in one place and automatically derive things from it.**

**This includes the migrations - unlike in Ruby On Rails, for example, migrations are entirely derived from your models file, and are essentially just a history that Django can roll through to update your database schema to match your current models**.