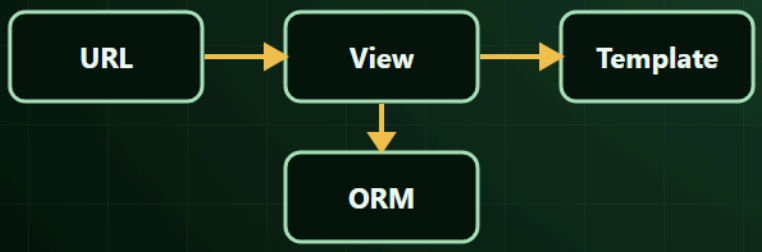
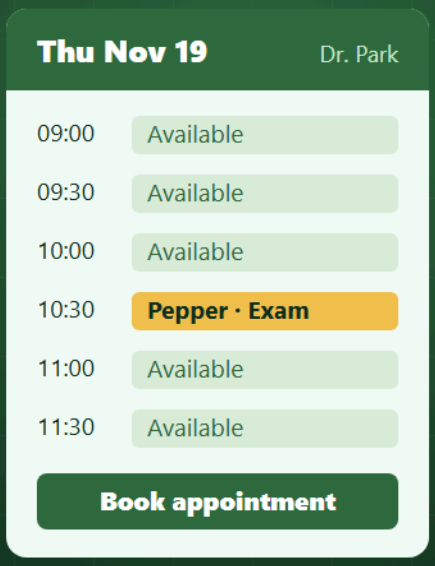
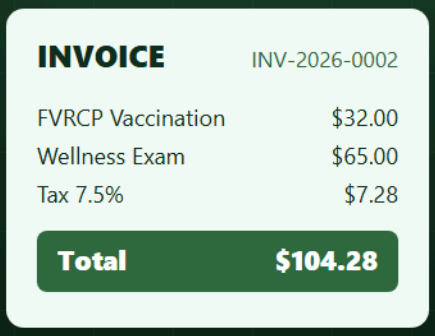
pets/models.py

```
class Pet(models.Model):
    owner = models.ForeignKey(
        Owner, related_name="pets")
    name = models.CharField(max_length=60)
    species = models.CharField(
        choices=Species.choices)
```




```
$ python manage.py migrate
Applying appointments.0001... OK
```





A COMPLETE, PRACTICAL GUIDE

Django and Python Web Development

Build real web apps from models to deployment

Models · ORM · Migrations · Admin · Forms · Views
Auth · Security · APIs · Async · Testing · Deployment

PYTHON · DJANGO 6 · POSTGRESQL · NGINX

THENEXBOOKS

Django and Python Web Development

Build Real Web Apps from Models to Deployment
From your first view to a tested, secured, production-ready application — built step by step

First Edition

THENEXBOOKS

Django and Python Web Development
Build Real Web Apps from Models to Deployment

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Django, Python, and the third-party packages discussed in this book change over time; settings, APIs, defaults, and supported versions can differ between releases. Always consult the official documentation and release notes for the versions you use, and test changes before deploying them.

Django is a registered trademark of the Django Software Foundation. Python is a registered trademark of the Python Software Foundation. PostgreSQL, SQLite, Nginx, Redis, Gunicorn, Uvicorn, Docker, GitHub, Linux, Windows, macOS, and other product names mentioned in this book are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Django Software Foundation, the Python Software Foundation, or any other organization named in it.

Sunmeadow Veterinary Clinic, the veterinary practice used as the running example in this book, is fictional, as are its vets, staff, pet owners, pets, appointments, medical notes, invoices, and all sample data. Medical details in the example are simplified for teaching and are not veterinary advice. The 7.5% tax rate and other business rules are invented for the example. Email addresses use reserved example domains, and phone numbers use the fictional 555 range.

Preface

Most useful software on the web looks surprisingly similar underneath. A person fills in a form, the server checks the input, saves some rows in a database, sends an email, and shows a page with the result. A clinic books appointments, a shop takes orders, a school enrolls students, a club manages members. The details change, but the shape stays the same: URLs, views, templates, data, users, and rules that must never be broken.

Django was built for exactly that shape of work. It gives you a URL router, a template language, an object-relational mapper with migrations, a ready-made admin back office, forms with validation, authentication and permissions, sessions, caching, testing tools, and sensible security defaults — all designed to work together and all written in Python. Instead of assembling a dozen libraries before you can show your first page, you start with a coherent toolkit and spend your time on the problem you actually want to solve.

This book teaches Django from the very beginning and takes you all the way to a deployed, production-ready application. We do not tour features in isolation. We build one real system step by step, introducing each part of Django at the moment it solves a problem we actually have, and explaining *why* it works the way it does, so you can design your own applications with confidence.

Who This Book Is For

- **Python beginners and intermediate programmers** who know the basics of the language and want to build real web applications with it.
- **Developers coming from other web frameworks** who want to understand how Django does routing, data access, forms, and authentication — and why.
- **Data and automation people** who already write Python scripts and need a proper web interface, database, and user accounts around them.
- **Students and self-taught developers** who want one complete, realistic project in their portfolio instead of a collection of disconnected snippets.

You should be comfortable with basic Python: variables, functions, lists and dictionaries, classes, and installing packages with `pip`. Chapter 1 includes a short refresher on the Python features Django leans on most. No prior web development experience is required.

What You Will Learn

- How the web works from Django's point of view: requests, responses, URLs, views, and templates.
- How to structure a project into focused apps, configure settings, and keep secrets out of your code.
- How to model real data with fields, relationships, choices, and constraints — and evolve it safely with migrations.

- How to query with the ORM, from simple filters to aggregation, annotations, subqueries, and fixing N+1 problems.
- How to give staff a back office in an afternoon with the Django admin.
- How to build forms, ModelForms, formsets, file uploads, and class-based views, and where business rules really belong.
- How to use sessions, cookies, messages, middleware, signals, management commands, and fixtures.
- How to add accounts, login, signup, groups, permissions, and object-level access control.
- How to protect an application against CSRF, XSS, clickjacking, and injection, including Content Security Policy.
- How to build JSON APIs with plain Django, and when Django REST Framework is worth adding.
- How to use async views, caching, and background tasks.
- How to test everything, configure each environment, and deploy with PostgreSQL, Gunicorn or Uvicorn, and Nginx.

How This Book Is Organized

Part I — Getting Started with Django (Chapters 1–5) sets up your environment and introduces projects, apps, settings, URLs, views, templates, and static files. **Part II — Data and the ORM** (Chapters 6–10) covers models, migrations, querying, advanced queries and performance, and the admin. **Part III — Building the Application** (Chapters 11–15) covers forms, ModelForms, formsets, file uploads, class-based views, the booking workflow with services and transactions, and signals, management commands, and fixtures. **Part IV — Users, Sessions, and Security** (Chapters 16–19) covers sessions, cookies, messages, middleware, authentication, permissions, and security. **Part V — APIs, Async, and Background Work** (Chapters 20–24) covers JSON APIs, API design, Django REST Framework, async views, caching, performance, and background tasks. **Part VI — Testing and Production** (Chapters 25–28) covers testing, configuration for every environment, deployment, and a complete launch case study. The appendices provide a cheat sheet, templates and reference material, a production checklist, and a glossary.

Each chapter ends with "Try It Yourself" exercises and a **Key Takeaways** box for quick review.

The Running Example: Sunmeadow Veterinary Clinic

Throughout the book we build the web application for **Sunmeadow Veterinary Clinic**, a fictional small-town practice with three vets — Dr. Elena Park in general practice, Dr. Marcus Reid for surgery and dentistry, and Dr. Priya Nair, who sees rabbits and birds — and a busy front desk. Pet owners such as Jordan Walker, whose golden retriever Biscuit loves treats and hates nail trims, want to book appointments online. The front desk needs to manage owners, pets, and the daily schedule. Vets need to write and sign visit notes. The clinic needs invoices with correct tax, late-cancellation fees, reminders, and reports.

We start with a single hello page and finish with a complete, tested application: an online booking system with real business rules, a staff back office, visit records, invoicing, a JSON API, background emails, caching, and a production deployment. The sample data is small enough to check by hand, and every shell session, query result, and test run printed in this book was produced by running the code against Django.

Versions and Conventions

Examples target **Django 6.x** on **Python 3.12** or newer, and were verified with Django 6.1. Django 6.0 and later require Python 3.12+. Where a feature needs a specific release — template partials, built-in Content Security Policy, and the tasks framework arrived in Django 6.0; the `MAILERS` email setting in Django 6.1 — a Note says so. Django also publishes long-term support (LTS) releases; check the official Django website for the current release schedule and support dates before you choose a version for a new project. Development uses SQLite, which ships with Python; production uses PostgreSQL. Third-party packages such as Django REST Framework, Gunicorn, Uvicorn, and WhiteNoise are always labeled as such; check their documentation for the versions that support your Django release.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, errors, or security problems.

Commands are shown with a `$` prompt, such as `$ python manage.py migrate`; inside an activated virtual environment they work the same on Windows, macOS, and Linux. Interactive Django shell sessions use the `>>>` prompt. Code, file names, and settings appear in `monospace`, and each listing is labeled with its file path relative to the project folder, such as `pets/models.py`. Long lines are wrapped to fit the page. The clinic's time zone is America/Chicago, and the sample data takes place in the week of November 16, 2026.

Contents

Preface

PART I — GETTING STARTED WITH DJANGO

1 Why Django? How Web Apps Work and Setting Up Your Environment

2 Your First Project: startproject, runserver, and a Hello Page

3 Projects and Apps: Structure, Settings, and the Sunmeadow Plan

4 URLs and Function-Based Views: Routing Requests to Python

5 Templates: The Django Template Language, Inheritance, and Static Files

PART II — DATA AND THE ORM

6 Models: Fields, Choices, Relationships, and Constraints

7 Migrations: Evolving Your Schema Safely

8 The ORM in Depth: QuerySets, Lookups, and Related Objects

9 Advanced Queries: Aggregation, Annotations, F and Q, and Query Performance

10 The Django Admin: A Back Office for the Front Desk

PART III — BUILDING THE APPLICATION

11 Forms: Validation, Rendering, and the Request Cycle

12 ModelForms, Formsets, and File Uploads

13 Class-Based Views and Generic Views

14 The Booking Workflow: Business Rules, Services, and Transactions

15 Signals, Management Commands, and Fixtures

PART IV — USERS, SESSIONS, AND SECURITY

16 Sessions, Cookies, Messages, and Middleware

17 Authentication: Custom Users, Login, Logout, and Signup

18 Permissions, Groups, and Protecting Views and Data

19 Security Essentials: CSRF, XSS, Clickjacking, CSP, and Safe Settings

PART V — APIS, ASYNC, AND BACKGROUND WORK

20 JSON APIs with Plain Django: JsonResponse, Serialization, and Validation

21 API Design in Practice: Tokens, Pagination, Errors, and CORS

22 Django REST Framework: An Overview of the Third-Party Option

23 Async Views, ASGI, and Real-Time Options

24 Caching, Performance, and Background Tasks

PART VI — TESTING AND PRODUCTION

25 Testing Django Apps: TestCase, the Test Client, and Test Data

26 Settings for Every Environment: Configuration, Logging, and Email

27 Deployment: PostgreSQL, Gunicorn or Uvicorn, Nginx, and Static Files

28 Case Study: Shipping Sunmeadow — Launch Checklist, Monitoring, and Next Steps

APPENDICES

A Django Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Getting Started with Django

How web applications work and where Django fits, setting up Python and a virtual environment, your first project and page, organizing Sunmeadow into focused apps, routing URLs to views, and rendering HTML with templates and static files.

1 · Why Django?

2 · Your First Project

3 · Projects and Apps

4 · URLs and Function-Based Views

5 · Templates and Static Files

CHAPTER 1 · PART I — GETTING STARTED WITH DJANGO

Why Django? How Web Apps Work and Setting Up Your Environment

Before we write a single view, we need a clear picture of what happens when someone opens a web page, what a framework like Django does for us, and a clean, repeatable Python setup to build on. By the end of this chapter you will have Django installed in a virtual environment, a Git repository ready for code, and a good idea of the clinic application we will build together.

What Happens When You Open a Web Page

Every web application, from a tiny personal blog to a large online store, follows the same basic conversation. A **client** (usually a browser, sometimes a phone app or another program) sends a **request** to a **server**. The server works out what the client wants, does some work, and sends back a **response**. Then the connection's job is done. The next click starts a brand-new conversation. That conversation is called **HTTP**, the Hypertext Transfer Protocol, and HTTPS is the same protocol wrapped in encryption.

Let's follow one real request through the system we are going to build. Jordan Walker owns a golden retriever named Biscuit. Jordan opens a browser on a Wednesday morning and visits the Sunmeadow Veterinary Clinic website to see which appointment slots are free on Thursday. The browser sends a request that looks roughly like this:

HTTP request (simplified)

```
GET /appointments/slots/?date=2026-11-19 HTTP/1.1
Host: sunmeadow.example.com
Accept: text/html
Cookie: sessionid=...
```

The first line carries three pieces of information. `GET` is the **method**: the client wants to read something, not change it. `/appointments/slots/` is the **path**, which names the resource. The part after the question mark, `?date=2026-11-19`, is the **query string**, a set of extra parameters. The lines after that are **headers**: the host name, the formats the browser accepts, and a cookie that lets the server recognize Jordan between requests.

The server answers with a response that has a **status code**, its own headers, and usually a **body**:

HTTP response (simplified)

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8

<!doctype html>
<html lang="en">
  <head><title>Available slots | Sunmeadow</title></head>
  <body>
    <h1>Thursday, November 19</h1>
    ...
  </body>
</html>
```

The browser reads the HTML, fetches any stylesheets and images it mentions (each one another request), and draws the page. When Jordan picks a slot and presses "Book", the browser sends a different kind of request, a `POST`, carrying the form data in its body. The server checks the data, stores a booking in its database, and answers, typically with a redirect that sends the browser to a confirmation page.

The pieces you will meet again and again

Concept	What it means	Sunmeadow example
Method	The kind of action requested	`GET` a page, `POST` a booking form
Path	Which resource the client wants	`/owners/3/`
Query string	Optional parameters after `?`	`?q=walker`
Status code	How the request went	200 OK, 302 redirect, 404 not found
Headers	Metadata on requests and responses	`Content-Type`, `Cookie`
Body	The payload	HTML page, form data, JSON

Status codes come in families. Codes in the 200s mean success, 300s mean "look somewhere else" (redirects), 400s mean the client asked for something wrong or forbidden (404 Not Found, 403 Forbidden, 405 Method Not Allowed), and 500s mean the server failed. You will see all of these in this book, and Django gives you a clean way to produce each one.

Where the database fits

A server that only sent fixed files would be a plain website. A **web application** remembers things: who the owners are, which pets they have, which appointment slots are taken, which invoices are unpaid. That memory lives in a **database**, usually a relational database such as SQLite (a single file, perfect for development) or PostgreSQL (a full database server, which we use in production in Chapter 27). For each request, the application reads from or writes to the database and then builds a response from the results.

One request, end to end

```
Browser                 Server running Django            Database
-------                 ---------------------            --------
GET /owners/3/  ----->  1. match the URL to a view
                        2. view asks for owner 3  ----->  SELECT ... WHERE id = 3
                                                  <-----  one row
                        3. fill an HTML template
200 OK + HTML   <-----  4. return the response
```

That four-step loop is the heart of almost every page in this book. The rest is detail: validating what users type, deciding who is allowed to see what, sending emails, keeping things fast, and running the whole thing safely on a real server.

What a Web Framework Does for You

You could write a web application with nothing but Python's standard library. You would parse raw HTTP, split the path by hand, open database connections, build SQL strings, glue HTML together with string concatenation, invent your own login system, and defend against every common attack yourself. People did exactly that in the early days of the web, and the results were usually slow to build and easy to break.

A **web framework** packages the solutions to these recurring problems so you can concentrate on what makes your application different. A full-featured framework typically gives you:

- **Routing**: mapping URLs such as `/owners/3/` to the Python function that handles them.
- **Request and response objects**: Python objects instead of raw bytes.
- **Templates**: a safe way to produce HTML from data.
- **Database access**: reading and writing rows through Python objects instead of hand-written SQL.
- **Forms**: rendering inputs, validating what comes back, and showing errors.
- **Authentication**: user accounts, password hashing, login and logout.
- **Security protections**: defenses against cross-site request forgery, cross-site scripting, SQL injection, and clickjacking.
- **Tooling**: a development server, a test runner, and commands for managing the database.

What Django Is

Django is a free, open-source web framework written in Python. It has been in production use since the mid-2000s (its first public release was in 2005), it is maintained by a large community under the Django Software Foundation, and it is widely used for everything from content sites to internal business tools. Django describes itself as "batteries included": instead of asking you to pick a separate library for every task, it ships with a coherent set of tools that are designed to work together.

The batteries we will use

- **The URL dispatcher** routes requests to views (Chapter 4).
- **The Django Template Language** turns data into HTML, escaping it safely by default (Chapter 5).
- **The ORM** (object-relational mapper) lets you define database tables as Python classes and query them with Python code (Chapters 6 to 9).
- **Migrations** keep your database schema in step with your models (Chapter 7).
- **The admin site** generates a back office for staff from your models, almost for free (Chapter 10).
- **Forms** validate input and render errors (Chapters 11 and 12).
- **Authentication, sessions, permissions, and messages** cover users and what they may do (Chapters 16 to 18).
- **Security middleware** handles CSRF protection, clickjacking headers, HTTPS settings, and, in Django 6.0 and later, a built-in Content Security Policy (Chapter 19).
- **Caching, email, background tasks, and async views** help the application grow (Chapters 23 and 24).
- **A test framework** built on Python's `unittest` (Chapter 25).

MTV: Django's name for a familiar idea

Many frameworks describe their structure as **MVC**: Model, View, Controller. Django uses the same separation of concerns but different words, usually summarized as **MTV**: Model, Template, View. The naming trips up newcomers, so here is the mapping:

Django term	Responsibility	Classic MVC term
Model	Data and the rules about it; one class per table	Model
Template	How data is presented as HTML	View
View	Handles a request: loads data, picks a template	Controller
URLconf + framework	Routes each request to the right view	Part of the controller

In Django, a "view" is the Python function or class that receives a request and returns a response. It decides *what* data to show; the template decides *how* it looks. Keep that distinction in mind and the rest of the vocabulary falls into place.

Security by default

One of Django's quiet strengths is that its defaults are safe. Templates escape HTML automatically, so a pet owner who types `<script>` into a notes field cannot inject code into a staff member's browser. Forms that change data require a CSRF token. The ORM passes values to the database as parameters

rather than pasting them into SQL strings. Passwords are stored as salted, slow hashes, never in plain text. You can still make mistakes, and Chapter 19 shows the ones to watch for, but you start from a protected position rather than an exposed one.

When Django Fits, and When Something Else Might

Django shines when your application has a real data model and many of the classic web features: user accounts, forms, an internal admin area, permissions, email, and HTML pages rendered on the server. A veterinary clinic with owners, pets, appointments, visit notes, and invoices is a textbook example. So are booking systems, school portals, inventory tools, content sites, and most of the internal tools businesses run on.

Python has other good web frameworks, and it is worth knowing where they differ:

- **Flask** is a small "micro" framework. It gives you routing, request handling, and templates, and leaves the database layer, forms, and authentication to extensions you choose yourself. That flexibility suits small services and people who want to assemble their own stack.
- **FastAPI** focuses on building JSON APIs, with request validation driven by Python type hints and automatic API documentation. It is a natural fit for API-only services.

None of these is "the best" in general; they make different trade-offs. Django trades some flexibility for a large set of integrated, well-documented parts. If you need an admin back office, accounts, and server-rendered pages, Django saves you from building or wiring those yourself. And as Part V shows, Django can also serve JSON APIs, run async views, and queue background tasks.

Django Versions and the Release Model

This book targets **Django 6.x** running on **Python 3.12 or newer**. Django 6.0 raised the minimum Python version to 3.12, so older interpreters will not work. All of the code in this book was run on Django 6.1 with Python 3.12, and where a feature first appeared in a specific release, we say so (for example, template partials arrived in 6.0, and the `MAILERS` email setting arrived in 6.1).

Django follows a predictable release pattern:

- **Feature releases** (such as 6.0 and 6.1) add new features and appear on a regular cycle.
- **Patch releases** (such as 6.1.1) fix bugs and security issues without changing behavior you depend on. Always install the latest patch release of your chosen feature release.
- **Long-term support (LTS) releases** are the x.2 releases, for example Django 5.2 LTS. They receive security fixes for a longer period, which suits teams who upgrade less often.

Django also has a **deprecation policy**: when a feature is going to be removed, it first emits deprecation warnings for a couple of feature releases. That gives you time to update your code. Chapter 28 shows how to surface those warnings in your test runs before an upgrade.

> **NOTE**
>
> Support periods change over time, so we do not print end-of-support dates in this book. Check the official Django download page, which includes the current release schedule and support status, before you choose a version for a new project.

The Python You Need for This Book

You do not need to be a Python expert, but Django code leans on a handful of language features. If any of the following look unfamiliar, spend a few minutes with them in a Python shell before you continue. Every snippet below was run with Python 3.12, and the outputs are exactly what Python printed.

Virtual environments and pip

A **virtual environment** is a private folder of installed packages for one project. Without one, every project on your computer shares the same packages, and upgrading Django for one project can break another. With one, each project gets its own Django, its own versions, and its own `pip`. We create the environment in a folder called `.venv` inside the project, a widely used convention that editors recognize automatically. `pip` is Python's package installer; we always run it as `python -m pip` so it is guaranteed to install into the Python that is currently active.

f-strings

An f-string is a string literal with an `f` prefix; expressions inside braces are evaluated and inserted. A format spec after a colon controls presentation, and a trailing `=` prints the expression and its value, which is handy for quick debugging.

Python shell

```
>>> pet = "Biscuit"
>>> weight = 31.2
>>> print(f"{pet} weighs {weight:.2f} kg")
Biscuit weighs 31.20 kg
>>> print(f"{pet=}")
pet='Biscuit'
```

Decimal for money

Binary floating-point numbers cannot represent most decimal fractions exactly. That is fine for physics, and wrong for invoices. Python's `decimal` module stores decimal numbers exactly and lets you choose a rounding rule. Sunmeadow stores every price and total as a `Decimal`, and Chapter 14 shows a real invoice that comes out one cent wrong with floats.

Python shell

```
>>> 0.1 + 0.2
0.30000000000000004
>>> from decimal import Decimal, ROUND_HALF_UP
```

```
>>> Decimal("0.1") + Decimal("0.2")
Decimal('0.3')
>>> price = Decimal("65.00")
>>> tax = (price * Decimal("0.075")).quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
>>> print(tax, price + tax)
4.88 69.88
```

Notice that we build decimals from **strings**, not floats. `Decimal(0.1)` would faithfully copy the float's tiny error. `quantize(Decimal("0.01"))` rounds to cents; `ROUND_HALF_UP` is the "round half away from zero" rule most people learned at school.

datetime, timedelta, and zoneinfo

Appointments happen at specific moments in a specific place. Python's `datetime` objects can be **naive** (no time zone) or **aware** (attached to a time zone). Django stores aware datetimes in UTC and converts them to local time for display, and the standard library's `zoneinfo` module supplies the time zone rules. Sunmeadow is in the `America/Chicago` time zone.

Python shell

```
>>> from datetime import date, datetime, timedelta
>>> from zoneinfo import ZoneInfo
>>> chicago = ZoneInfo("America/Chicago")
>>> start = datetime(2026, 11, 19, 10, 30, tzinfo=chicago)
>>> start.isoformat()
'2026-11-19T10:30:00-06:00'
>>> start.astimezone(ZoneInfo("UTC")).isoformat()
'2026-11-19T16:30:00+00:00'
>>> print(start + timedelta(minutes=30))
2026-11-19 11:00:00-06:00
>>> (date(2026, 11, 18) - date(2019, 4, 12)).days
2777
```

Thursday 10:30 in Chicago in November is 16:30 UTC, because central standard time is six hours behind UTC. A `timedelta` represents a duration; adding 30 minutes gives the end of a Sunmeadow appointment slot. Subtracting two dates gives a `timedelta` too, whose `.days` tells us Biscuit has been alive for 2,777 days on the story's Wednesday.

> **NOTE**
>
> On Windows, Python has no built-in time zone database, so `zoneinfo` relies on the `tzdata` package. Installing Django on Windows pulls in `tzdata` automatically, so you normally do not have to think about it.

Decorators

A decorator is a function that takes a function and returns a new function, usually one that adds behavior before or after the original. The `@name` line above a definition is shorthand for `greet = log_call(greet)`. Django uses decorators everywhere: `@login_required`, `@require_POST`,

`@admin.register`, `@receiver`, and `@task` all appear later in this book.

decorators.py

```
import functools


def log_call(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} {kwargs}")
        return func(*args, **kwargs)
    return wrapper


@log_call
def greet(name, *, clinic="Sunmeadow"):
    return f"Welcome to {clinic}, {name}!"


print(greet("Jordan"))
print(greet("Aisha", clinic="Sunmeadow Veterinary Clinic"))
```

Terminal

```
$ python decorators.py
Calling greet with ('Jordan',) {}
Welcome to Sunmeadow, Jordan!
Calling greet with ('Aisha',) {'clinic': 'Sunmeadow Veterinary Clinic'}
Welcome to Sunmeadow Veterinary Clinic, Aisha!
```

The bare `*` in `greet(name, *, clinic=...)` makes every parameter after it **keyword-only**: callers must write `clinic="..."` explicitly. Sunmeadow's service functions use this style, for example `book_slot(*, owner, pet, slot, service, reason)`, so that a call can never mix up the pet and the owner by position.

*args and **kwargs

`*args` collects extra positional arguments into a tuple, and `**kwargs` collects extra keyword arguments into a dictionary. The same symbols work in reverse at a call site: `**info` unpacks a dictionary into keyword arguments. Django's ORM is built around this: `Pet.objects.filter(species="cat")` receives `species` as a keyword argument, and you can build filters dynamically from a dictionary.

Python shell

```
>>> def describe_pet(name, **details):
...     parts = [f"{key}={value}" for key, value in details.items()]
...     return f"{name}: " + ", ".join(parts)
...
>>> describe_pet("Luna", species="cat", breed="Siamese")
'Luna: species=cat, breed=Siamese'
```

```
>>> info = {"species": "rabbit", "breed": "Holland Lop"}
>>> describe_pet("Clover", **info)
'Clover: species=rabbit, breed=Holland Lop'
```

Classes and __str__

Django models, forms, and class-based views are ordinary Python classes. You will define attributes, write methods, inherit from Django base classes, and override methods such as `__str__`, which controls how an object prints. Compare the two outputs below: `print()` uses `__str__`, while a list shows each element's `repr()`, which falls back to a default with a memory address (yours will differ). Django models get a friendlier default repr, as you will see in Chapter 8.

Python shell

```
>>> class Pet:
...     def __init__(self, name, species):
...         self.name = name
...         self.species = species
...     def __str__(self):
...         return f"{self.name} ({self.species.title()})"
...
>>> print(Pet("Biscuit", "dog"))
Biscuit (Dog)
>>> [Pet("Pepper", "cat")]
[<__main__.Pet object at 0x000001E1E13FBF80>]
```

If you are comfortable reading all of the snippets above, you have enough Python for the whole book. Anything else we need, we explain where it appears.

Installing Python

You need Python 3.12 or newer. Check what you have first:

Terminal

```
$ python --version
Python 3.12.0
```

Any 3.12.x or later version is fine; your patch number will likely be newer than the one shown. If the command is missing or reports an older version, install a current Python:

- **Windows:** download the installer from python.org. It installs the **`py` launcher**, which lets you pick a version explicitly: `py -3.12 --version`. On Windows, `py` is often more reliable than `python` outside a virtual environment, because the `python` command may be intercepted by an app-store alias or point at a different installation.
- **macOS:** use the python.org installer or a package manager you already trust. Use `python3` if `python` is not available.

- **Linux:** use your distribution's packages (for example `python3.12` and its `venv` package) or a tool you already use for managing Python versions.

The exact installer does not matter. What matters is that you can run a Python 3.12+ interpreter and create a virtual environment with it.

Creating the Project Folder and Virtual Environment

Pick a place for your code, create a folder for the clinic project, and create a virtual environment inside it. We call the folder `sunmeadow`; in Chapter 2, Django will fill it with the project files.

Terminal

```
$ mkdir sunmeadow
$ cd sunmeadow
$ python -m venv .venv
```

On Windows, if `python` is not found, use `py -3.12 -m venv .venv` instead. Now **activate** the environment. Activation adjusts your shell's `PATH` so that `python` and `pip` refer to the copies inside `.venv`:

Terminal

```
# macOS and Linux
$ source .venv/bin/activate

# Windows, Command Prompt
> .venv\Scripts\activate

# Windows, PowerShell
> .venv\Scripts\Activate.ps1
```

Your prompt changes to show `(.venv)` at the start. From now on, **inside an activated virtual environment, `python` works on every operating system**, so the book always writes commands as `$ python ...`. If you open a new terminal window, activate the environment again before running project commands. To leave the environment, run `deactivate`.

> **WARNING**
>
> If PowerShell refuses to run `Activate.ps1` with a message about execution policies, do not weaken your system's security settings globally just to make it work. Use the Command Prompt activation script instead, or ask whoever manages your machine. Many editors also activate the environment automatically when you open the project folder.

A brand-new environment contains only `pip` itself:

Terminal

```
(.venv) $ python -m pip list
Package Version
------- -------
pip     23.2.1
```

Your pip version will differ, and pip may print a notice suggesting an upgrade. Upgrading pip inside the environment with `python -m pip install --upgrade pip` is harmless and often a good idea.

Installing Django

Install the current 6.x release of Django into the environment. The version specifier in quotes says "6.1 or newer, but below 6.2", which gets you the latest bug-fix release of the 6.1 series and nothing that might change behavior underneath you:

Terminal

```
(.venv) $ python -m pip install "Django>=6.1,<6.2"
```

The quotes matter: without them, many shells treat `>` and `<` as redirection operators. pip downloads Django and its small set of dependencies and finishes with a line beginning `Successfully installed` that lists Django, `asgiref`, and `sqlparse`, plus `tzdata` on Windows. Then confirm the installation:

Terminal

```
(.venv) $ python -m django --version
6.1.1
```

You may see a later 6.1.x number, which is exactly what we want. You can also ask from Python itself:

Python shell

```
>>> import django
>>> django.get_version()
'6.1.1'
>>> django.VERSION
(6, 1, 1, 'final', 0)
```

> **TIP**
>
> Record your dependencies from day one. Create a file called `requirements.txt` containing the single line `Django>=6.1,<6.2`. Anyone (including you, on a new laptop) can then recreate the environment with `python -m pip install -r requirements.txt`. Chapter 27 shows how to pin exact versions for production.