Get Started With Django Part 1: Build a Portfolio App

by Jasmine Finer Apr 01, 2019 79 Comments **django** **intermediate** **web-de**

Table of Contents

257

- Why You Should Learn Django
- The Structure of a Django Website
- What You're Going to Build
- Hello, World!
  - Set Up Your Development Environment
  - Create a Django Project
  - Create a Django Application
  - Create a View
  - Add Bootstrap to Your App
- Showcase Your Projects
  - Projects App: Models
  - Projects App: Views
  - Projects App: Templates
- Share Your Knowledge With a Blog
  - Blog App: Models
  - Blog App: Django Admin
  - Blog App: Views
  - Blog App: Templates
- Conclusion

68

Django is a fully featured Python web framework that can be used to build co
web applications. In this tutorial, you'll jump in and learn Django by example
follow the steps to create a fully functioning web application and, along the w
learn some of the most important features of the framework and how they w
together.

There are endless web development frameworks out there, so why should yo
Django over any of the others? First of all, it's written in Python, one of the m
readable and beginner-friendly programming languages out there.

> **Note:** This tutorial assumes an intermediate knowledge of the Python language. If you're new to programming with Python, check out some of our beginner tutorials or the introductory course.

The second reason you should learn Django is the scope of its features. If you
to build a website, you don't need to rely on any external libraries or package
choose Django. This
and the syntax is sea

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


There's also the add
library or framework

If you do find yourse
libraries that you ca

One of the great thin
It has detailed docu
examples and even

There's also a fantas
almost always a way forward by either checking the docs or asking the comm

Django is a high-level web application framework with loads of features. It's
for anyone new to web development due to its fantastic documentation, and
particularly if you're also familiar with Python.

The Structure of a Django Website

A Django website consists of a single **project** that is split into separate **apps**.
idea is that each app handles a self-contained function that the site needs to
perform. As an example, imagine an application like Instagram. There are sev
different functions that need to be performed:

- **User management**: Login, logout, register, and so on
- **The image feed**: Uploading, editing, and displaying images
- **Private messaging**: Private messages between users and notifications

These are each separate pieces of functionality, so if this were a Django site,
each piece of functionality should be a different Django app inside a single D
project.

In Django, the architecture is slightly different. Although based upon the MVC pattern, Django handles the controller part itself. There's no need to define h database and views interact. It's all done for you!

The pattern Django utilizes is called the Model-View-Template (MVT) pattern. view and template in the MVT pattern make up the view in the MVC pattern. A need to do is add some URL configurations to map the views to, and Django the rest!

A Django site starts off as a project and is built up with a number of applicati that each handle separate functionality. Each app follows the Model-View-Te pattern. Now that you're familiar with the structure of a Django site, let's ha at what you're going

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


What You'r

Before you get start
up with a plan of wh
application with the

- **A fully functio**
  a blog is a grea
  update, and de
  sort them. Finally, users will be able to leave comments on posts.

- **A portfolio of your work**: You can showcase previous web developmentprojects here. You'll build a gallery style page with clickab to projects that you've completed.

> **Note:** Before you get started, you can pull down the source code and follo along with the tutorial.
>
> If you prefer to follow along by writing the code yourself, don't worry. I've referenced the relevant parts of the source code throughout so you can re back to it.

We won't be using any external Python libraries in this tutorial. One of the gr things about Django is that it has so many features that you don't need to rel external libraries. However, we will add Bootstrap 4 styling in the templates.

By building these two apps, you'll learn the basics of Django models, view fu forms, templates, and the Django admin page. With knowledge of these featu you'll be able to go away and build loads more applications. You'll also have tools to learn even more and build sophisticated Django sites.

```
$ python3 -m venv venv
```

This command will create a folder `venv` in your working directory. Inside this directory, you'll find several files including a copy of the Python standard libr Later, when you install new dependencies, they will also be stored in this dir Next, you need to activate the virtual environment by running the following command:

```
$ source venv/bin/activate
```


Note: If you're nc command to acti need this comma

```
C:\> venv\Scr
```


You'll know that you prompt in the terminal will change. It should look something like this:

```
(venv) $
```

Note: Your virtual environment directory doesn't have to be called `venv`. I want to create one under a different name, for example `my_venv`, just repl with the second `venv` with `my_venv`.

Then, when activating your virtual environment, replace `venv` with `my_venv`again. The prompt will also now be prefixed with `(my_venv)`.

Now that you've created a virtual environment, it's time to install Django. Yo do this using `pip`:

```
(venv) $ pip install Django
```


Once you've set up the virtual environment and installed Django, you can no

```
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   │
│   └── manage.py
│
└── venv/
```

Most of the work you do will be in that first `personal_portfolio` directory. T
having to `cd` through several directories each time you come to work on your
project, it can be he
While you're in the r

```
$ mv personal_pc
$ mv personal_pc
$ rm -r personal
```

You should end up w

```
rp-portfolio/
│
├── personal_portfolio/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
├── venv/
│
└── manage.py
```

Once your file structure is set up, you can now start the server and check tha
set up was successful. In the console, run the following command:

```
$ python manage.py runserver
```

Then, in your browser go to `localhost:8000`, and you should see the followi

Create a Django Application

For this part of the tutorial, we'll create an app called `hello_world`, which yo
subsequently delete as its not necessary for our personal portfolio site.

To create the app, run the following command:

```
$ python manage.py startapp hello_world
```

This will create another directory called `hello_world` with several files:

- **`__init__.py`** t
- **`admin.py`** cont
- **`apps.py`** conta
- **`models.py`** con tables.
- **`tests.py`** cont
- **`views.py`** cont the HTML temp

Once you've created
`portfolio/setting`

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'hello_world',
]
```

That line of code means that your project now knows that the app you just c
exists. The next step is to create a view so that you can display something to

Create a View

Views in Django are a collection of functions or classes inside the `views.py` fi
your app directory. Each function or class handles the logic that gets process
each time a different URL is visited.

called `templates` inside your app directory. Create that directory and subsec
a file named `hello_world.html` inside it:

```
$ mkdir hello_world/templates/
$ touch hello_world/templates/hello_world.html
```

Add the following lines of HTML to your file:

```
<h1>Hello, World!</h1>
```


You've now created
user. The final step i
created. Your projec
URL configuration fo
add the following:

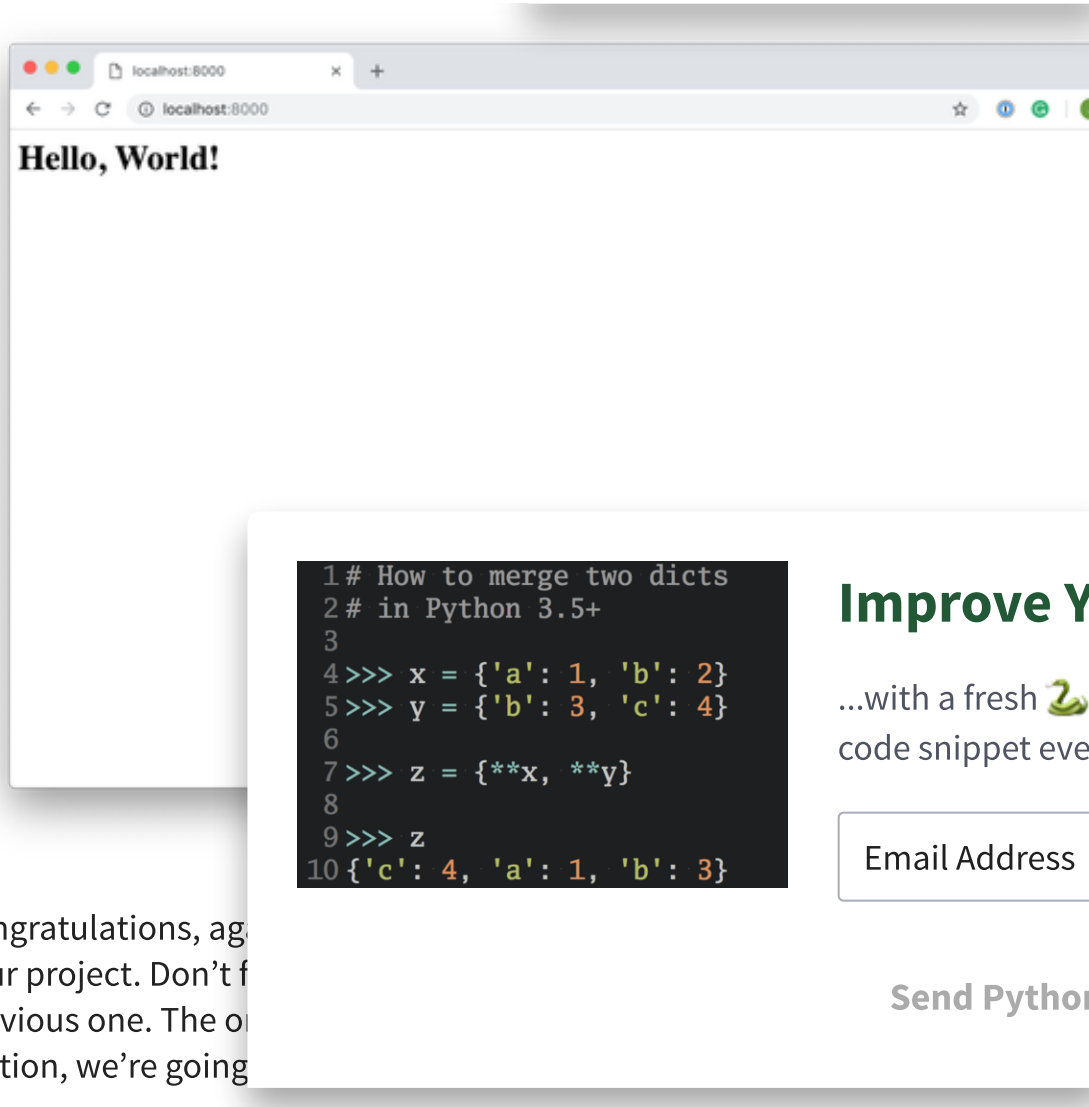
```
from django.cont
from django.urls

urlpatterns = [
    path('admin/
    path('', include('hello_world.urls')),
]
```

This looks for a module called `urls.py` inside the `hello_world` application a
registers any URLs defined there. Whenever you visit the root path of your UR
(`localhost:8000`), the `hello_world` application's URLs will be registered.
The `hello_world.urls` module doesn't exist yet, so you'll need to create it:

```
$ touch hello_world/urls.py
```

Inside this module, we need to import the path object as well as our
app's `views`module. Then we want to create a list of URL patterns that corres
to the various view functions. At the moment, we have only created one view
function, so we need only create one URL:

```
from django.urls import path
from hello_world import views
```



Congratulations, ag
your project. Don't f
previous one. The o
section, we're going

Add Bootstrap to Your App

If you don't add any styling, then the app you create isn't going to look too n
Instead of going into CSS styling with this tutorial, we'll just cover how to add
bootstrap styles to your project. This will allow us to improve the look of the
without too much effort.

Before we get started with the Bootstrap styles, we'll create a base template
can import to each subsequent view. This template is where we'll subsequen
the Bootstrap style imports.

Create another directory called `templates`, this time inside `personal_portf`
and a file called `base.html`, inside the new directory:

```
$ mkdir personal_portfolio/templates/
$ touch personal_portfolio/templates/base.html
```

We create this additional templates directory to store HTML templates that w

```
{% block page_content %}
<h1>Hello, World!</h1>
{% endblock %}
```

What happens here is that any HTML inside the `page_content` block gets add
inside the same block in `base.html`.

To install Bootstrap in your app, you'll use the Bootstrap CDN. This is a really
way to install Bootstrap that just involves adding a few lines of code to `base.`
Check out the source code to see how to add the CDN links to your project.

All future templates
Bootstrap styling on

Before we can see o
that `base.html` exis
app, but not in the p
update `TEMPLATES`:

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Improve Y

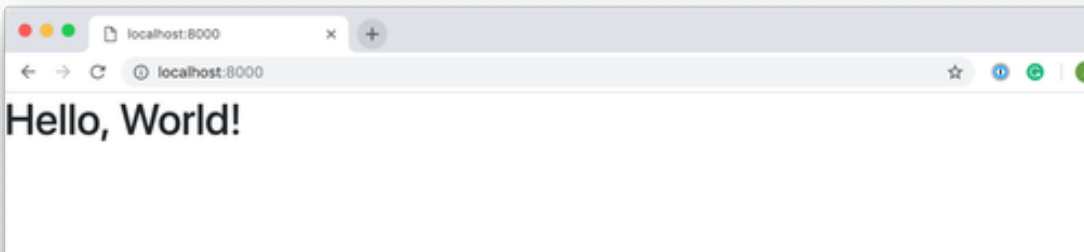
...with a fresh
code snippet eve

Email Address

Send Pythor

```
TEMPLATES = [
    {
        "BACKEND
        "DIRS":
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.mess
            ]
        },
    }
]
```

Now, when you visit `localhost:8000`, you should see that the page has been
formatted with slightly different styling:



In this section, you learned how to create a simple *Hello, World!* Django site b
creating a project with a single app. In the next section, you'll create another
application to showcase web development projects, and you'll learn all abou
models in Django!

The source code for this section can be found on GitHub.

Showcase Your Projects

Any web developer looking to create a portfolio needs a way to show off proj
they have worked on. That's what you'll be building now. You'll create anoth
Django app called p
displayed to the use
your work.

Before we build the
you need to do is de
line `"hello_world"`

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```



```
INSTALLED_APPS =
    'django.cont
    'django.cont
    'django.cont
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'hello_world',  # Delete this line
]
```

Finally, you need to remove the URL path created in `personal_portfolio/u`

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('hello_world.urls')),  # Delete this line
]
```

Now that you've removed the `hello_world` app, we can create the `projects`
Making sure you're in the `rp-portfolio` directory, run the following comma
your console:

Projects App: Models

If you want to store data to display on a website, then you'll need a database
Typically, if you want to create a database with tables and columns within th
tables, you'll need to use SQL to manage the database. But when you use Dja
you don't need to learn a new language because it has a built-in Object Relat
Mapper (ORM).

An ORM is a program that allows you to create classes that correspond to dat
tables. Class attributes correspond to columns, and instances of the classes
correspond to rows in the database. So, instead of learning a whole new lang
to create our databa

When you're using a
referred to as **mode**
app.

In your projects app
display to the user.

The model you'll cre

- **title** will be a
- **description** w
- **technology** will be a string field, but its contents will be limited to a set number of choices.
- **image** will be an image field that holds the file path where the image is

To create this model, we'll create a new class in `models.py` and add the follo
our fields:

```
from django.db import models

class Project(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    technology = models.CharField(max_length=20)
    image = models.FilePathField(path="/img")
```

Django models come with many built-in model field types. We've only used t
this model. `CharField` is used for short strings and specifies a maximum leng

`TextField` is similar to `CharField` but can be used for longer form text as it c
have a maximum length limit. Finally, `FilePathField` also holds a string but
point to a file path name.

Now that you've create a migration file, you need to apply the migrations set
the migrations file and create your database using the `migrate` command:

```
$ python manage.py migrate projects
Operations to perform:
  Apply all migrations: projects
Running migrations:
  Applying projects.0001_initial... OK
```

Note: When runn
added `projects`
migrations in the
created.

If you run `makemi`
migrations for all
and applied. This
are not needed.

You should also see
your project. Now y
in your table that ar

To create instances of our `Project` class, we're going to have to use the Djan
shell. The Django shell is similar to the Python shell but allows you to access
database and create entries. To access the Django shell, we use another Djar
management command:

```
$ python manage.py shell
```

Once you've accessed the shell, you'll notice that the command prompt will
from `$` to `>>>`. You can then import your models:

```
>>> from projects.models import Project
```

We're first going to create a new project with the following attributes:

- **name:** `My First Project`
- **description:** `A web development project.`
- **technology:** `Django`

```
...     technology='Flask',
...     image='img/project2.png'
... )
>>> p2.save()
>>> p3 = Project(
...     title='My Third Project',
...     description='A final development project.',
...     technology='Django',
...     image='img/project3.png'
... )
>>> p3.save()
```

Well done for reachi
in Django and build
into database tables
your model class.

In the next section,
function to display t
this section of the tu

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Projects App:

Now you've created the projects to display on your portfolio site, you'll need
create view functions to send the data from the database to the HTML templ

In the `projects` app, you'll create two different views:

1. An index view that shows a snippet of information about each project
2. A detail view that shows more information on a particular topic

Let's start with the index view, as the logic is slightly simpler. Inside `views.py`
need to import the `Project` class from `models.py` and create a
function `project_index()` that renders a template called `project_index.ht`
the body of this function, you'll make a Django ORM query to select all objec
the `Project`table:

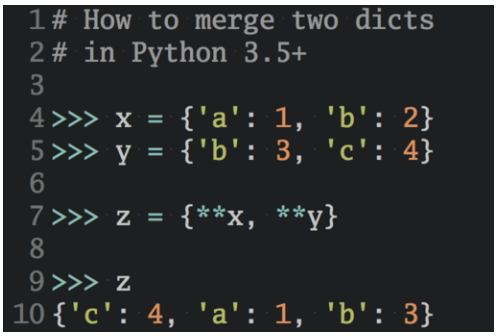
```
from django.shortcuts import render
from projects.models import Project

def project_index(request):
    projects = Project.objects.all()
    context = {
        'projects': projects
    }
```

We also render a template named `project_index.html`, which doesn't exist
Don't worry about that for now. You'll create the templates for these views in
next section.

Next, you'll need to create the `project_detail()` view function. This functio
need an additional argument: the id of the project that's being viewed.

Otherwise, the logic is similar:

```
13 def project_detail(request, pk):
14     project = Project.objects.get(pk=pk)
15     context
16         'pro
17     }
18     return r
```



In **line 14**, we perfor
key, pk, equal to tha
our `context` diction
template `project_c`

Once your view func
by creating a file `pro`
file should contain the following code:

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path("", views.project_index, name="project_index"),
6     path("<int:pk>/", views.project_detail, name="project_de
7 ]
```

In **line 5**, we hook up the root URL of our app to the `project_index` view. It is
slightly more complicated to hook up the `project_detail` view. To do this, w
the URL to be `/1`, or `/2`, and so on, depending on the `pk` of the project.

The `pk` value in the URL is the same `pk` passed to the view function, so you ne
dynamically generate these URLs depending on which project you want to vi
do this, we used the `<int:pk>` notation. This just tells Django that the value p
in the URL is an integer, and its variable name is `pk`.

With those now set up, we need to hook these URLs up to the project URLs.
In `personal_portfolio/urls.py`, add the following highlighted line of code

Projects App: Templates

Phew! You're nearly there with this app. Our final step is to create two templa

1. The `project_index` template
2. The `project_detail` template

As we've added Bootstrap styles to our application, we can use some pre-sty
components to make the views look nice. Let's start with
the `project_index`template.

For the `project_index` template, you'll create a grid of Bootstrap cards, with
card displaying deta
there are going to b

We don't want to ha
the information to e
template engine: **fo**

Using this feature, y
for each one. The fo

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Improve Y

...with a fresh 🐍
code snippet eve

Email Address

Send Python

```
{% for project i
{# Do something.
{% endfor %}
```

Now that you know how for loops work, you can add the following code to a
named `projects/templates/project_index.html`:

```
{% extends "base.html" %}
{% load static %}
{% block page_content %}
<h1>Projects</h1>
<div class="row">
{% for project in projects %}
    <div class="col-md-4">
        <div class="card mb-2">
            <img class="card-img-top" src="{% static project
            <div class="card-body">
                <h5 class="card-title">{{ project.title }}</
                <p class="card-text">{{ project.description
                <a href="{% url 'project_detail' project.pk
                   class="btn btn-primary">
                    Read More
                </a>
```

Django automatically registers static files stored in a directory named `static` each application. Our image file path names were of the structure: `img/<photo_name>.png`.

When loading static files, Django looks in the `static/` directory for files matc given filepath within `static/`. So, we need to create a directory named `stat` another directory named `img/` inside. Inside `img/`, you can copy over the ima from the source code on GitHub.

On **line 6**, we begin the for loop, looping over all projects passed in by the `context`dictionary.

Inside this for loop, attributes, you can access the project's used to access any c

On **line 9**, we includ `static project.im` file matching `proje`

The final point that our `project_detai` files. The code for th

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Improve Y

...with a fresh 🐍 code snippet eve

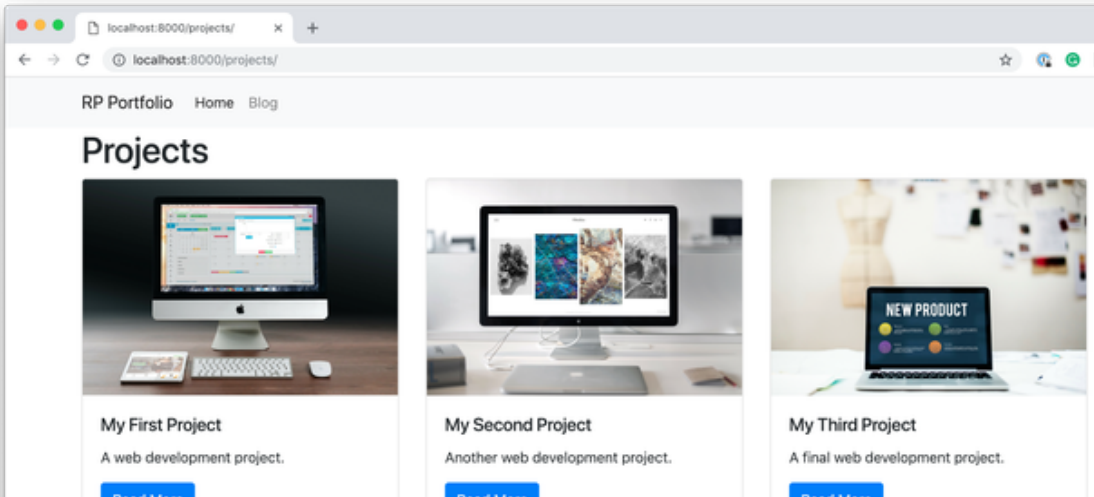
Email Address

Send Pythor

```
{% url '<url path name>' <view_function_arguments> %}
```

In this case, we are accessing a URL path named `project_detail`, which tak integer arguments corresponding to the `pk` number of the project.

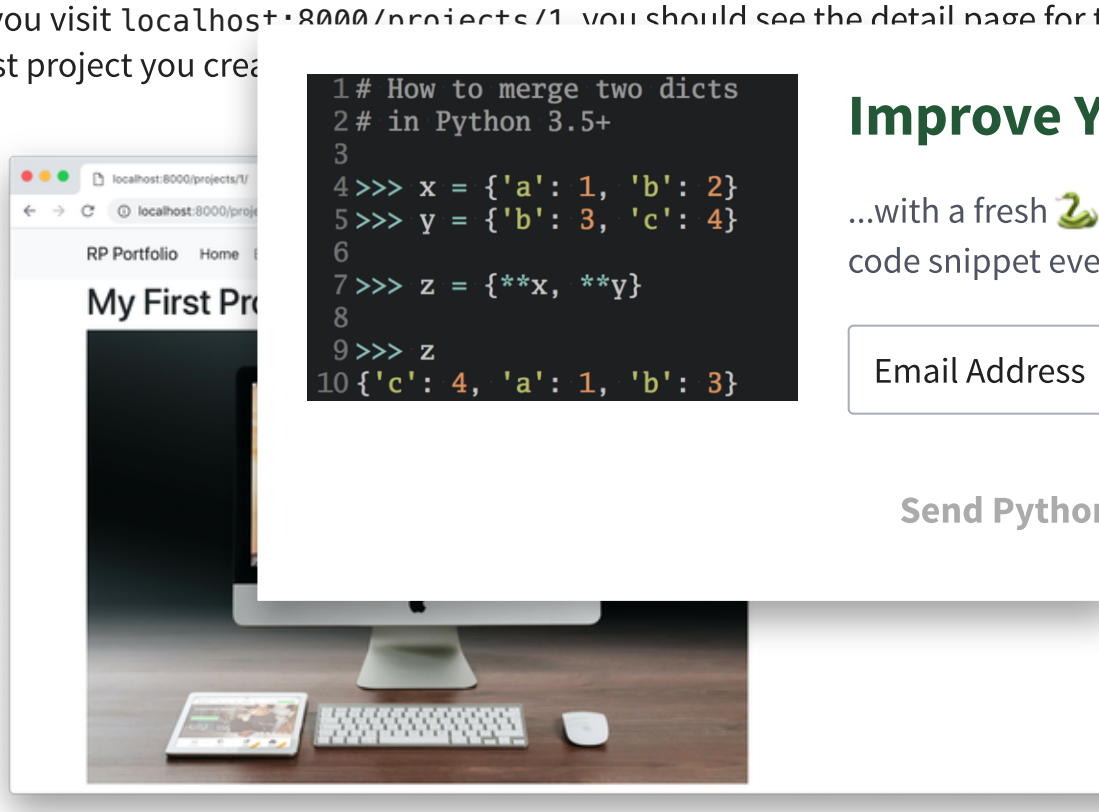
With all that in place, if you start the Django server and visit `localhost:8000/projects`, then you should see something like this:



```
        <h5>Technology used:</h5>
        <p>{{ project.technology }}</p>
    </div>
</div>
{% endblock %}
```

The code in this template has the same functionality as each project card in the `project_index.html` template. The only difference is the introduction o Bootstrap columns.

If you visit `localhost:8000/projects/1` you should see the detail page for t first project you crea



In this section, you learned how to use models, views, and templates to crea fully functioning app for your personal portfolio project. Check out the sourc codefor this section on GitHub.

In the next section, you'll build a fully functioning blog for your site, and you' learn about the Django admin page and forms.

Share Your Knowledge With a Blog

A blog is a great addition to any personal portfolio site. Whether you update monthly or weekly, it's a great place to share your knowledge as you learn. In section, you're going to build a fully functioning blog that will allow you to p the following tasks:

```
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "projects",
    "blog",
]
```

Hold off on hooking up the URLs for now. As with the `projects` app, you'll sta
adding your models.

Blog App: Mo

The `models.py` file i

You're going to neec

1. `Post`
2. `Category`
3. `Comment`

These tables need to
models come with fields specifically for this purpose.

Below is the code for the `Category` and `Post` models:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=20)

class Post(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
    created_on = models.DateTimeField(auto_now_add=True)
    last_modified = models.DateTimeField(auto_now=True)
    categories = models.ManyToManyField('Category', related_
```

The `Category` model is very simple. All that's needed is a single `CharField` in
we store the name of the category.

The `title` and `body` fields on the `Post` model are the same field types as you
the `Project` model. We only need a `CharField` for the `title` as we only want

relationship from a `Category` object, even though we haven't added a field t
adding a `related_name` of `posts`, we can access `category.posts` to give us a
posts with that category.

The third and final model we need to add is `Comment`. We'll use another relati
field similar the `ManyToManyField` that relates `Post` and `Category`. However,
only want the relationship to go one way: one post should have many comm

You'll see how this works after we define the `Comment` class:

```
16 class Comment(models.Model):
17     author =
18     body = m
19     created_
20     post = m
```


The first three fields
users to add a name
a `created_on` field t

On **line 20**, we use a
the `ManyToManyFie`
reasoning behind th
can't have a comment that corresponds to many posts.

The `ForeignKey` field takes two arguments. The first is the other model in the
relationship, in this case, `Post`. The second tells Django what to do when a p
deleted. If a post is deleted, then we don't want the comments related to it h
around. We, therefore, want to delete them as well, so we add the
argument `on_delete=models.CASCADE`.

Once you've created the models, you can create the migration files
with `makemigrations`:

```
$ python manage.py makemigrations blog
```

The final step is to migrate the tables. This time, don't add the app-specific fl
Later on, you'll need the `User` model that Django creates for you:

```
$ python manage.py migrate
```


Now that you've created the models, we can start to add some posts and cat

Before you can access the admin, you need to add yourself as a superuser. Th
why, in the previous section, you applied migrations project-wide as oppose
for the app. Django comes with built-in user models and a user management
that will allow you to login to the admin.

To start off, you can add yourself as superuser using the following command

```
$ python manage.py createsuperuser
```

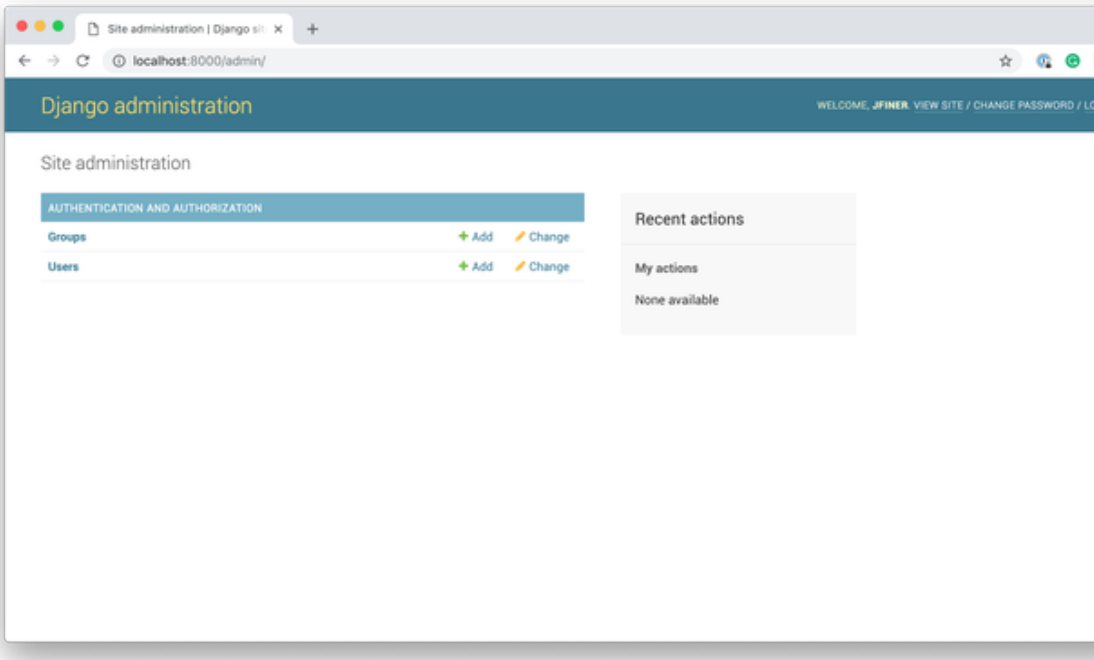
You'll then be prompted to enter a username followed by your email address
password. Once you
superuser has been
start again:

```
Username (leave
Email address: j
Password:
Password (again)
Superuser create
```



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Navigate to `localho`
create a superuser. You'll see a page similar to the one below:

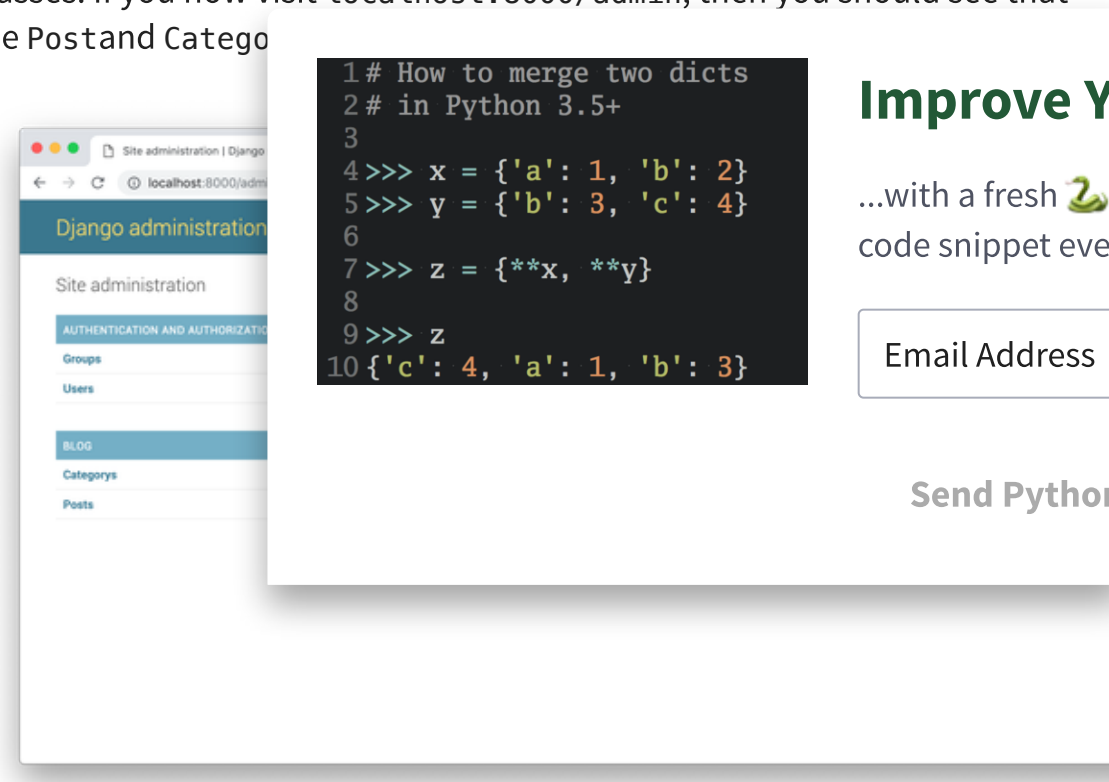


The User and Groups models should appear, but you'll notice that there's no
reference to the models you've created yourself. That's because you need to
them inside the admin.

> If you wanted to add a feature where comments are moderated, then go ahead and add the Comments model too. The steps to do so are exactly th same!

On **line 4** and **line 7**, you define empty classes `PostAdmin` and `CategoryAdmi` the purposes of this tutorial, you don't need to add any attributes or methoc these classes. They are used to customize what is shown on the admin pages this tutorial, the default configuration is enough.

The last two lines are the most important. These register the models with the classes. If you now visit `localhost:8000/admin`, then you should see that the `Post`and `Catego`



If you click into *Posts* or *Categorys*, you should be able to add new instances models. I like to add the text of fake blog posts by using lorem ipsum dummy

Create a couple of fake posts and assign them fake categories before moving the next section. That way, you'll have posts you can view when we create ou templates.

Don't forget to check out the source code for this section before moving on to building out the views for our app.

Blog App: Views

You'll need to create three view functions in the `views.py` file in the `blog` dire

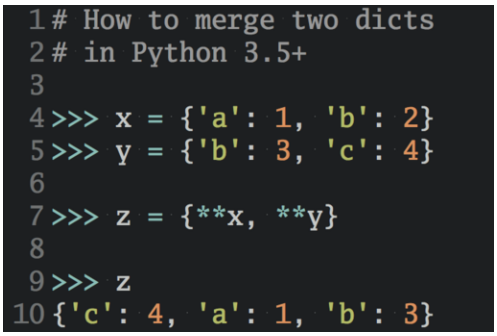
- **`blog_index`** will display a list of all your posts.

On **line 2**, you import the `Post` model, and on **line 5** inside the view function, obtain a Queryset containing all the posts in the database. `order_by()` order Queryset according to the argument given. The minus sign tells Django to sta the largest value rather than the smallest. We use this, as we want the posts ordered with the most recent post first.

Finally, you define the `context` dictionary and render the template. Don't wo about creating it yet. You'll get to creating those in the next section.

Next, you can start to create the `blog_category()` view. The view function w to take a category name as an argument and query the `Post` database for all that have been assigned the given category:

```
13 def blog_cat
14     posts =
15         cate
16     ).order_
17         '-cr
18     )
19     context
20         "cat
21         "pos
22     }
23     return r
```



On **line 14**, you've used a Django Queryset filter. The argument of the filter te Django what conditions need to be met for an object to be retrieved. In this c only want posts whose categories contain the category with the name corresponding to that given in the argument of the view function. Again, you using `order_by()` to order posts starting with the most recent.

We then add these posts and the category to the `context` dictionary, and ren template.

The last view function to add is `blog_detail()`. This is more complicated as going to include a form. Before you add the form, just set up the view functio show a specific post with a comment associated with it. This function will be equivalent to the `project_detail()` view function in the `projects` app:

```
21 def blog_detail(request, pk):
22     post = Post.objects.get(pk=pk)
23     comments = Comment.objects.filter(post=post)
24     context = {
25         "post": post,
26         "comments": comments,
```

> **Note:** If the `CharField` of your form corresponds to a model `CharField`, m
> sure both have the same `max_length` value.

`blog/forms.py` should contain the following code:

```
from django import forms

class CommentForm(forms.Form):
    author = forms.CharField(
        max_length=60,
        widget=forms.TextInput(attrs={
            "cla
            "pla
        })
    )
    body = forms
        attrs={
            "cla
            "pla
        })
    )
```


You'll also notice an
The `author` field has the `forms.TextInput` widget. This tells Django to load t
as an HTML text input element in the templates. The `body` field uses
a `forms.TextArea`widget instead, so that the field is rendered as an HTML te
element.

These widgets also take an argument `attrs`, which is a dictionary and allows
specify some CSS classes, which will help with formatting the template for th
later. It also allows us to add some placeholder text.

When a form is posted, a `POST` request is sent to the server. So, in the view fu
we need to check if a `POST` request has been received. We can then create a
comment from the form fields. Django comes with a handy `is_valid()` on it
so we can check that all the fields have been entered correctly.

Once you've created the comment from the form, you'll need to save it
using `save()`and then query the database for all the comments assigned to t
given post. Your view function should contain the following code:

```
def blog_detail(request, pk):
    post = Post.objects.get(pk=pk)

    form = CommentForm()
```

```
from . import CommentForm
```

We then go on to check if a `POST` request has been received. If it has, then we
a new instance of our form, populated with the data entered into the form.

The form is then validated using `is_valid()`. If the form is valid, a new instar
of `Comment` is created. You can access the data from the form
using `form.cleaned_data`, which is a dictionary.

They keys of the dictionary correspond to the form fields, so you can access t
author using `form.c`
to the comment wh

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Improve Y

...with a fresh
code snippet eve

Email Address

Send Pythor

Note: The life cyc
an outline of how

1. When a use
the server. I
want to ren
2. When a use
a `POST`reque
the server. A
happen:
    - The form is valid, and the user is redirected to the next page.
    - The form is invalid, and empty form is once again displayed. Th
user is back at step 1, and the process repeats.

The Django forms module will output some errors, which you can display to
user. This is beyond the scope of this tutorial, but you can read more
about rendering form error messages in the Django documentation.

On **line 33**, save the comment and go on to add the `form` to the `context` dict
so you can access the form in the HTML template.

The final step before you get to create the templates and actually see this blo
and running is to hook up the URLs. You'll need create another `urls.py` file
inside `blog/` and add the URLs for the three views:

```
from django.urls import path
from . import views

urlpatterns = [
```

- **`localhost:8000/blog`:** Blog index
- **`localhost:8000/blog/1`:** Blog detail view of blog with `pk=1`
- **`localhost:8000/blog/python`:** Blog index view of all posts with category `python`

These URLs won't work just yet as you still need to create the templates.

In this section, you created all the views for your blog application. You learne
to use filters when making queries and how to create Django forms. It won't
now until you can see your blog app in action!

As always, don't forg
GitHub.

Blog App: Ter

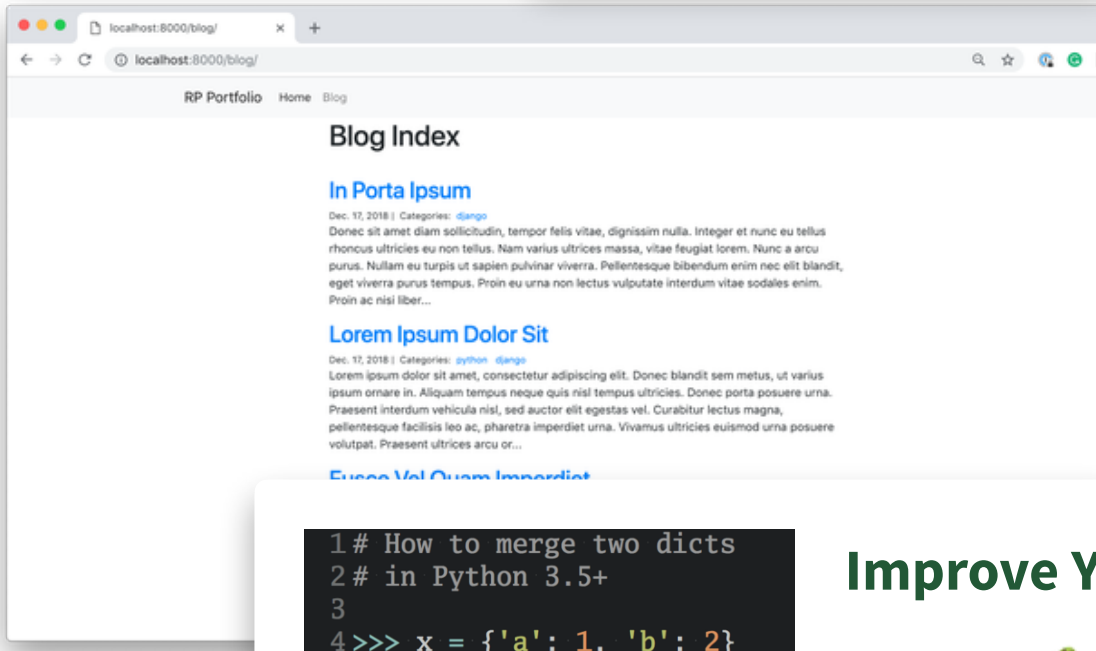
The final piece of ou
have created a fully

You'll notice there a
the interface prettie
they do but do chec

The first template y
file `blog/templates/blog_index.html`. This will be very similar to the proje
index view.

You'll use a for loop to loop over all the posts. For each post, you'll display th
and a snippet of the body. As always, you'll extend the base
template `personal_porfolio/templates/base.html`, which contains our
navigation bar and some extra formatting:

```
{% extends "base.html" %}
{% block page_content %}
<div class="col-md-8 offset-md-2">
    <h1>Blog Index</h1>
    <hr>
    {% for post in posts %}
    <h2><a href="{% url 'blog_detail' post.pk%}">{{ post.tit
    <small>
        {{ post.created_on.date }} | 
        Categories: 
        {% for category in post.categories.all %}
        <a href="{% url 'blog_category' category.name %}">
            {{ category.name }}
        </a> 
```

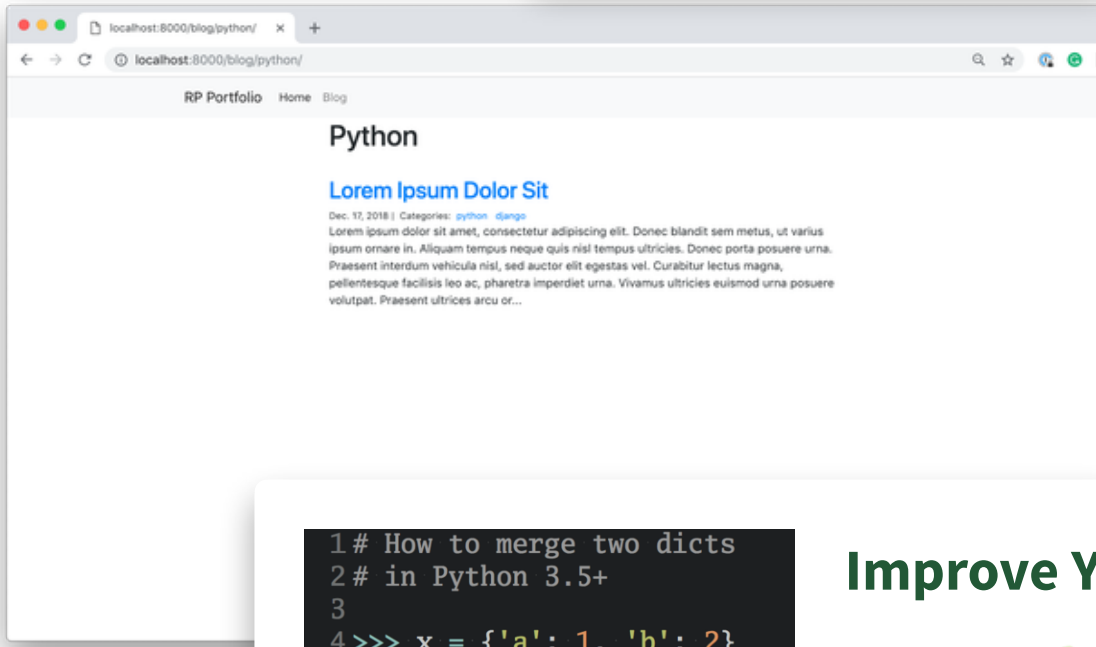


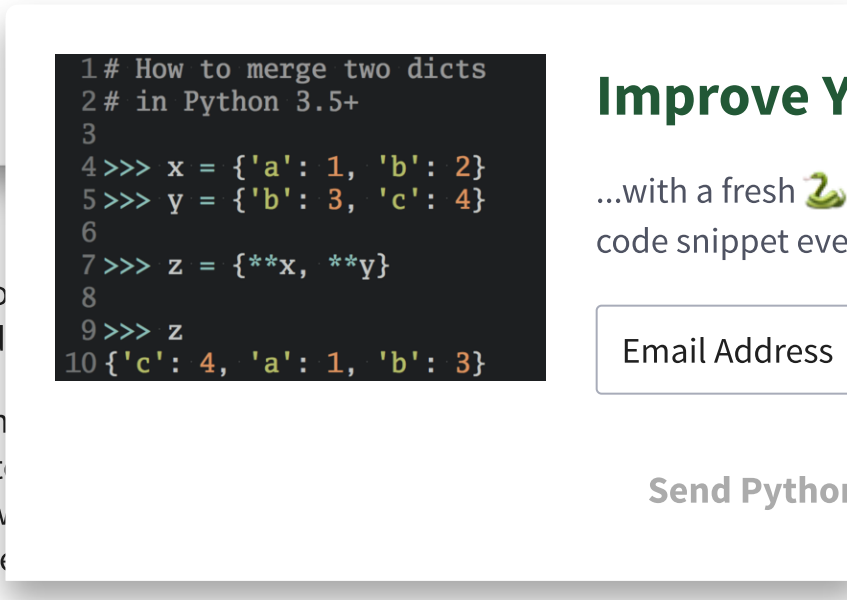

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Next, create another
your `blog_category`
to `blog_index.html`
of `Blog Index`:

```
1  {% extends '
2  {% block page_content %}
3  <div class="col-md-8 offset-md-2">
4      <h1>{{ category | title }}</h1>
5      <hr>
6      {% for post in posts %}
7          <h2><a href="{% url 'blog_detail' post.pk%}">{{ post
8          <small>
9              {{ post.created_on.date }} | 
10             Categories: 
11             {% for category in post.categories.all %}
12             <a href="{% url 'blog_category' category.name %}
13                 {{ category.name }}
14             </a> 
15             {% endfor %}
16         </small>
17         <p>{{ post.body | slice:":400" }}...</p>
18     {% endfor %}
19 </div>
20 {% endblock %}
```

Most of this template is identical to the previous template. The only differenc
on **line 4**, where we use another Django template filter `title`. This applies ti





257

The last template to
display the title and

68

Between the title an
created and any cat
users can add a new
have already been le

```
{% extends "base.html" %}
{% block page_content %}
<div class="col-md-8 offset-md-2">
    <h1>{{ post.title }}</h1>
    <small>
        {{ post.created_on.date }} | 
        Categories: 
        {% for category in post.categories.all %}
        <a href="{% url 'blog_category' category.name %}">
            {{ category.name }}
        </a> 
        {% endfor %}
    </small>
    <p>{{ post.body | linebreaks }}</p>
    <h3>Leave a comment:</h3>
    <form action="/blog/{{ post.pk }}/" method="post">
        {% csrf_token %}
        <div class="form-group">
            {{ form.author }}
        </div>
        <div class="form-group">
```

Underneath the post, on **line 16**, you'll display your form. The form action po
the URL path of the page to which you're sending the `POST` request to. In this
it's the same as the page that is currently being visited. You then add a `csrf_`
which provides security and renders the body and author fields of the form,
followed by a submit button.

To get the bootstrap styling on the author and body fields, you need to add
the `form-control` class to the text inputs.

Because Django renders the inputs for you when you include `{{ form.body }}` and `{{ form.author }}`, you can't add these classes in the template. Tha
you added the attribu

Underneath the forr
the given post. The
displayed.

Once that template
visit `localhost:800`

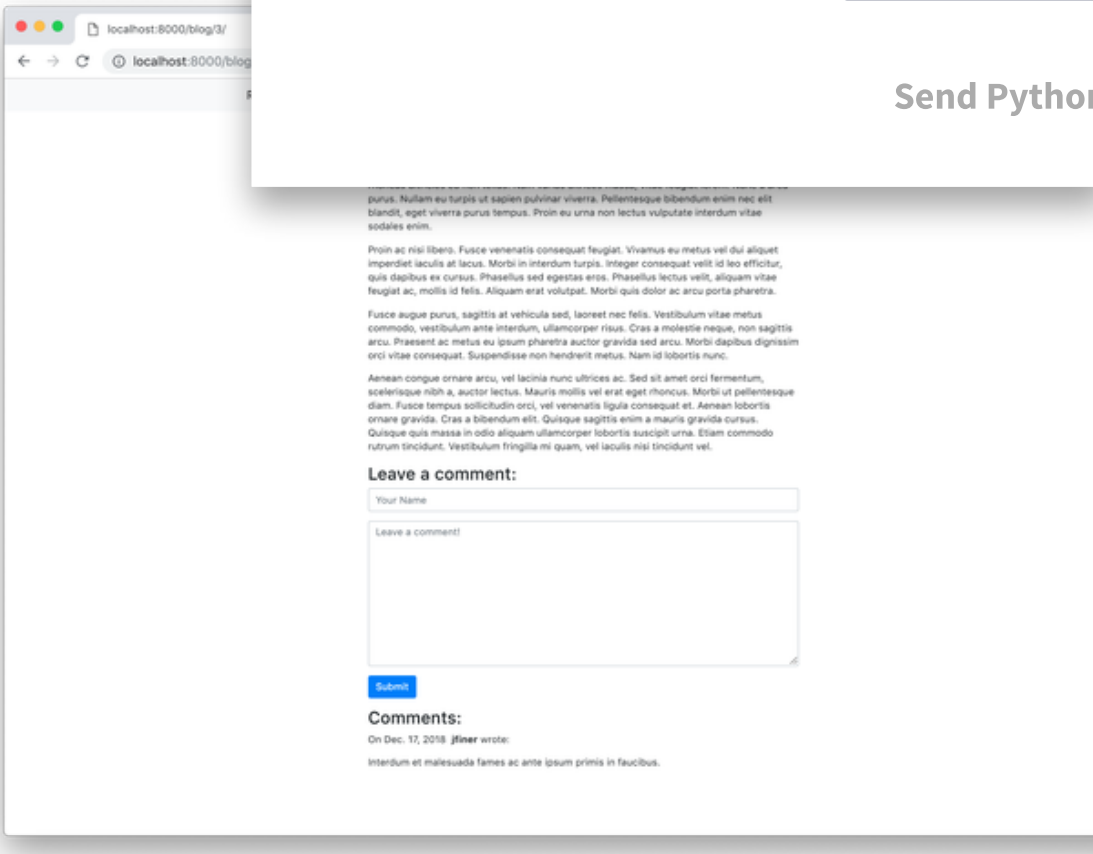
```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Improve Y

...with a fresh
code snippet eve

Email Address

Send Pythor



You should also be able to access the post detail pages by clicking on their tit
the `blog_index` view.

The final finishing touch is to add a link to the `blog_index` to the navigation
in `base.html`. This way, when you click on *Blog* in the navigation bar, you'll h