Try Django

About
Dig into the bedrock of a Django app — models, templates, and views — and build one of your own along the way.

Getting Started (Level 1)

It is recommended that you have some experience with Python, HTML and database concepts.

Before anything, lets talk about Django is. Django is an open-source MVC (Model View Controller) framework, written in Python for building web applications. Django includes functionality to handle common web development tasks in an effort to make it easier to build Web apps with less code.

Like most web applications today, Django sends data to a server to validate, process it, and render HTML. Django is what's being run on the server to handle the validation, processing, and rendering. To describe it as a series of steps:

1. A user make a request for a URL, such as codeschool.com/learn
2. The request is sent to Django to be processed
3. All of the user information is looked up and validated
4. The data is passed off to Django for rendering
5. HTML is sent back to the user

During this process, Django will separate these responsibilities into separate components, which makes projects earlier to maintain and collaborate on.

In Django, the validation, rendering, and processing are taken care of by the following separate components: the model, template, and view (MTV). The View (controller) processes the user data, The Model stores and validates data, The Template (view) renders the HTML to the user. The model, template, and view are what make up the Django app.

During this course, we will be creating an application to keep track of various foods and treats. There will be an image, a name, the cost, and the location it was purchased. Make sure you have Django installed to follow along with the notes/challenges/questions for this course. Assuming you have Python installed, its as simple as entering "pip install django" in your terminal.

We will be writing the following coding conventions in this course:

> console commands	Commands that are entered into the console will be designated by a single carrot
>>> django shell commands	Commands that are entered into the django shell will be designated by three carrots
script_name.py	Any code that is written in Django script files will be designated with a file name, and then Python code

But before we start any projects or challenges, its important to understand the difference between projects and apps in Django. You could have a Project, that houses all the settings and files related to the project itself. Inside the project, you have Apps such as "blog" and "careers" (blog.project.com and careers.project.com) that have their own directory and related files.

Create a Django Project `> django-admin startproject <name>`	This command line command will create a Django project with whatever name you want the project to have <name>. This will set up the file directory and basic files needed to run the project: settings.py - Will hold the Project's Settings urls.py - Wil hold the Project's URLs manage.py - Is a utility for administrative tasks

Run a Django Project `> python manage.py runserver`	This command line command will start our project up, and run it locally on your machine. You will be told where to navigate to see your project. Ex. Starting development server at http://127.0.0.1:8000/ It is important to note that you need to be in the Project's directory to run this command.

Creating and App in a Project

```
> python manage.py startapp <name>
```

This command line command will create an app with name <name> inside a Project. This command will instantiate several files for the app, as well as its own directory.

Adding a view to an App

File: <project>/<app>/views.py

```
1 from django shortcuts import render
2 from django http import HttpResponse
3
4 # create custom views here
5 def <view>(request):
6   return HttpResponse('<h1>Hello Collectors!</h1>')
```

Note that this file is in the App directory. This is the file where we will be adding the view functions for the application.

1. We'll need to import render to render the templates
2. We also need HttpResponse to return a HTTP Response
5. Here we define a view function that takes in a web request
6. We'll return an HTTP Response, with an HTML parameter

Mapping a URL

File: <project>/<project>/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from <app-name> import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    #localhost/<view>
    url(r'^<view>/',
        views.index),
]
```

Before we can navigate to a new app, we first need to map a URL to it. Django has what's known as a URL Dispatcher, that maps a URL to a view in the project. To map a URL to a view, you need to add a new url object to the urlpatterns array. The first parameter uses REGEX to match the path being passed in. The second parameter is the view function. If the pattern passed in is matched, the request will be sent to the view in the app.

Refactoring URL Dispatcher

File: <project>/<project>/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from <app-name> import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    #localhost/8000 will go here
    url(r'^',
        views.index),
]
```

Before we move any further, we're going to cover best practices when dealing with the URL Dispatcher. If you wanted the home page of your application to be http://localhost:8000/index, it would be a nuisance to need to add the /index when entering the URL.
If we pass in an empty string with a carat to the REGEX, and navigate to http://localhost:8000 we will be redirected to http://localhost:8000/index, without needed to enter it.

It is also good practice to have a URL Dispatcher for the project, as well as a URL Dispatcher for an app.

Refactoring URL Dispatcher

File: <project>/<project>/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    #localhost/8000 will go here
    url(r'^',
        include('<app>.urls')),
]
```

File: <project>/<app>/urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
        url(r'^$', views.<view>),
]
```

Including the app's urls.py file in the project's urls.py will specify to the URL Dispatcher that we want to go to a view in an app specifically.

Templates (Level 2)

Now that we have a basic page working, we'll get started on adding UI, which we will accomplish through use of a template. Remember that we want to render a page for the use, we're sending in a view from an app. A template is just a view that is dynamically filled with data when it is rendered.

Registering Your App:

Note that you'll need to make a "templates" directory in an app. Django knows to look

File: <project>/<project>/settings.py
```
INSTALLED_APPS = [
    '<app>',
    'django.contrib.admin',
    'django.contrib.auth',
    ...
]
```

for that directory to find templates to use.

Additionally, you need to register your app in the project settings file. You simple add your app to the list of installed apps that is in the file. This will ensure that Django knows where to look for things like template folders.

Creating a Template:

File: <project>/<app>/templates/<template>.html
```
<!DOCTYPE html>
<html>
  <head>
    <title>FoodTracker</title>
  </head>
  <body>
    <h1>FoodTracker</h1>
  </body>
</html>
```

Note that this is only an example of a template, and not what they always have to look like.

The template to the left will be an HTML file that renders when it is returned to the user. Like other HTML files, we can add whatever tags we want to be displayed upon rendering the page.

Rendering a Template:

File: <project>/<app>/views.py
```
from django.shortcuts import render

def <view>(request):
    return render(request, '<template>')
```

When we want to render a template, we need only return a render object in the views.py file of an app. The render object will take in a request parameter, as well as the string name of the template we want to render.

Mentioned earlier, we can dynamically fill a template that is rendered. So we're going to start talking about how to do that. The first main step is sending data from the view to the template before it is rendered. The second step is to access the data that is now in the template.

Passing Dynamic Data to Template:

File: <project>/<app>/views.py
```
from django.shortcuts import render

def <view>(request):
    name = "Red Apple"
    value = "1.49"
    context = {"food_name": name,
               "food_value": value}
    return render(request, '<template>', context)
```

In the example to the left, we are setting data values of name and value. We then move the data into a dictionary object, which holds key-value pairs. When we return a render object, we'll pass it a third parameter of the dictionary object we just made, which is holding the data.

Accessing Dynamic Data in Template:

File: <project>/<app>/templates/<template>.html
```
<!DOCTYPE html>
<html>
  head>
    <title>FoodTracker</title>
  </head>
  <body>
    <h1>Food Tracker</h1>
    <p>{{ food_name }}</p>
    <p>{{ food_value }}</p>
  </body>
</html>
```

Again, this is just an example of a template in action.

When we pass in dynamic data to a template, we need to know how to access it. By using of the double curly brackets {{}}, we are able to inject the key-values into the HTML. This is a part of the Django Template Language.

It is important to remember the names of your key-value pairs to be able to inject properly.

Upon rendering, the key names in the template will be replaced with the value pairs that were passed in.

Now that we know how to send dynamic data to a template, let's look at what more we can do with that. If say, we wanted to be able to send more complex data (more than 2 field), or perhaps several data objects we'll need to get more creative. We could make a class of a data object to define information about the object, then have a list of data objects we want. After that, you just have to specify which object in the list we want to pass to the template.

Creating a Class and adding it to a view:

File: <project>/<app>/views.py
```
from django.shortcuts import render
```

As usual, this is just an example, to be used to show how to set up a class.
We start by creating a class, and a

```

def <view>(request):
    return render(request, "<template>", {"foods": foods})
class Food:
    def __init__(self, name, value, flavor, location):
        self.name = name
        self.value = value
        self.flavor = flavor
        self.location = location
foods = [
    Food("Apple", 1.99, "Sour", "Jim's Orchard")
    Food("Banana", 0.29, "Sweet", "Sam's Grove")
]
```

constructor (__init__) for our data object. This will be used to define the object holding our data.
We'll then make a list (foods) of our class objects, passing in values to initialize them with.
When we return the render object, we'll pass our list as a key-value pair in a dictionary object, so we can reference it later.

Django Language For Tags in a Template:

File: <project>/<app>/templates/<template>.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>FoodTracker</title>
  <head>
  <body>
    <h1>FoodTacker</h1>
    {% for food in foods %}
      <p>{{ food.name }}</p>
      <p>{{ food.value }}</p>
    {% endfor %}
  </body>
</html>
```

In this example, we'll look at the Django Language For Tag.
Now that we've passed in a list of class objects, we'll need a way to access them to display them in the template. Normally, in python, you could just write a for loop to go through every object in an iterable.
We'll be applying the same concept to our template. Django offers tags that you can use in a template. A Django tag starts and end with '%'.
We'll use a for tag here to iterate through our list. We'll just the "for index in iterable" syntax to loop through the list.
We can then use dot notation to reference members of our object to display in our template.
We then have to specify, using Django tags, that we're done with our loop.

Django Language If/Else Tags in a Template

File: <project>/<app>/templates/<template>.html

```
...
  <body>
    <h1>FoodTracker</h1>
    {% for food in foods %}
      <p>{{ food.name }}</p>
      {% if food.value > 0 %}
        <p>{{ food.value }}</p>
      {% else %}
        <p>Unknown</p>
      {% endif %}
    {% endfor %}
  </body>
<html>
```

In this continuation of the last example, we'll look the Django Language If/Else Tag.
When we go to display our data, we might want to display certain things under certain conditions. Now in python, or really any language, you could just use an if-else statement. The Django Language has this capability as well.
Again, we'll specify that we're using a Django tag with '%'. Then we just use "if <condition>" syntax to validate the if statement. Then use "else" to specify what to do if the if statement returns false. Then we'll end the if-else statement with "endif".
Do note, like regular if-else statements, if the conditional returns true, then only what's inside that will be rendered. Likewise, if the conditional returns false, only what's inside the else statement will render.

At this point, we are rendering dynamic data on our pages; but there is something we are missing... styles. We're only rendering HTML in our web app right now, so we'll need to add some styles to make everything look better.

Adding Static Files

File: <project>/<app>/static/styles.css

```
h1 {
  color: green;
}
```

When we want to add a styles sheet to an app, we'll need to make a "static" directory in the app and put it there.

Loading Static Files in Templates

File: <project>/<app>/templates/<templates>.html

```
{% load staticfiles %}

<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" type="text/css"
     href="{% static "styles.css" %}" />
  <title>FoodTracker</title>
```

When we want to include a styles sheet in our template, we'll first need to use the Django Template Language tag "load" to load static files.
This will let us link to the .css file using an HTML link tag in the head.
We'll still need to use Django Template Language to specify the static file as well as the name of it.
This will tell Django where to look to find the styles we want to include in the template.

```
</head>
<body>
...
```

Note that you can add Bootstrap to your project. You'll just need to add a boostrap.min.css file to your app's static directory, then link to it in your template like you would any other .css file.

Adding Bootstrap Elements to Our Template

File: <project>/<app>/templates/<template>.html

```
...
<body>
    <nav class="navbar navbar-default text-center">
        <a href="/">
        <img src="{% static 'images/<image>' %}">
        </a></nav>
</body>
...
```

This example that you have bootstrap in your app, linked up properly.
To add CSS elements to your template, you need only add them in an element's class like normal HTML.
For images, you want to first add an "images" directory inside your static directory. Then you can just call to the image in the directory using Django Template Language as seen to the left.

Note that you can add tables, containers, and other elements in HTML to your Django app. The only thing you'll need to do is make sure to use Django Template Language to inject and reference and render all of your elements properly.

Models (Level 3)

Now that we've covered adding dynamic data to a template, we'll want to look at a Model. In Django, a model defines the data's structure and communicates with a database. Being able to separate your data from the web app is important to having your app run efficiently, while easily being able to modify and access your data.

Creating a Data Model

File: <project>/<app>/models/<model>.py

```
from django.db import models

class <model>(models.Model):
    name = models.CharField()
    value = models.DecimalField()
    location = models.CharField()
    ...
```

When you create a model in Django, you mostly be following the class from the views file as a guide.
You'll starting by bring in models from Django, so we can actually make a model.
We'll then create a model that inherits from models.Model, so that Django knows its dealing with a model.
When populate the model with attributes that follow model data types.

Django Model Field Type	Python	SQL
models.CharField()	string	VARCHAR
models.IntegerField()	int	INTEGER
models.FloatField()	float	FLOAT
models.DecimalField()	decimal	DECIMAL

When we want to interact with a database, we need to specify our fields using Django Model Field Types. Each Model Field Type corresponds to a data type in both Python and SQL, so we can interact between the two.

There are many more Django Model Field Types, follow the link for reference:
https://docs.djangoproject.com/en/1.9/ref/models/fields/

Updating a Data Model

File: <project>/<app>/models/<model>.py

```
from django.db import models

class <model>(models.Model):
    name = models.CharField(max_length=100)
    value = models.DecimalField(max_digits=10,
                                decimal_places=2)
    location = models.CharField(max_length=100)
    ...
```

We'll need to specify attributes in our Django Model Field Types, otherwise we'll get errors when we try to run our app.
We specify the attributes we want when be declare the field type in a model.

Lets clarify a bit what a model is in the scope we've discussed so far. When we create and model an object in Django, we're in fact creating a database table and entries, but without needing to write SQL. We'll instead be using Django ORM (Object Relational Mapping), to translate Python code to SQL for us.

Perform a Migration

```
> python mangage.py makemigrations

> python manage.py migrate
```

Before we can add objects to our model, we'll need to make a migration, which is a sort of a version control system for the database.
The first command will make a migration file.
The second command will apply the migration to the database.
The steps are separated so you can review the migration before you run it.

Preview SQL in Migration

```
> python manage.py sqlmigrate <app> <migration>
```

If we run this command, we will get a preview the SQL commands in the migration.
Note that this step isn't necessary.

If you want to make sure that you are up to date, can run some of the commands that we ran earlier.
> python manage.py makemigrations will detect if there are any changes, and
> python manage migrate will confirm if there are any migrations to apply.

Now that we've made a migration, we can start up the Django Interactive Shell to interact with our objects through the shell.

Django Interactive Shell

```
> python manage.py shell
```

This command line command will open the Django shell. The shell is a good place to work our writing queries and making models before you do it in Python.

Django Shell: Import Models

```
>>> from <app>.models import <model>
```

Before we can start interacting with models, we need to import them to the shell. We'll do this the same way we would in the Python Interpreter.

Django Shell: Query Model

```
>>> <model>.objects.all()
```

This Django Shell query will pull all the objects that are in our model.

Retrieving Objects with a QuerySet

Django Shell	SQL	Note
<model>.objects	SELECT	QuerySet in Django equates to SELECT statement in SQL.
<model>.objects.all()	SELECT * FROM <model>	How we would return a list of all object in a model in Django vs SQL.
<model>.objects.filter(<field> = <value>)	SQL WHERE or LIMIT	We can filter objects in Django, similarly to filtering objects in SQL.
<model>objects.get(<field> = <value>)		If you know only 1 object will be returned, you can use the .get() operation, which is similar to using a primary key in SQL.

When we want to retrieve objects from a model in Django, we'll be using QuerySet. Now a Query set just represents a collection of objects from the database.

Django Shell: Creating Objects

```
>>> f = Food(name="Apple", value="1.29", location="Big Jim's Resturant", ...)
>>> f.save()
```

The first Django shell command will create an object. You'll need to follow the model details that were given to Django when you pass in parameters to the shell.
The second Django shell command will save the object to the database.

Adding a Descriptive __str__ method

File: <project>/<app>/models/<model>.py

```
from django.db import models

class <model>(models.Model):
    name = models.CharField(max_length=100)
    value = models.DecimalField(max_digits=10,
                                decimal_places=2)
    location = models.CharField(max_length=100)
    ...
    def __str__(self):
        return self.name
```

After saving our Food object to the database, we want to query to make sure it was save, so we run:

```
>>> Food.objects.all()
```

and we are returned with:

```
[<Food: Food object>]
```

But that's not very helpful, luckily there is a way to make the QuerySet more descriptive. We go into our model file, and add a __str__ method, that returns a parameter of the model.
That way, when we query our database again, we'll have something like this returned:

```
[Apple]
```

Updating View to Use Models

File: <project>/<app>/views.py

```python
from django.shortcuts import render
from .models import <model>

def <view>(request):
    # Get all the Model's objects
    objects = <model>.objects.all()
    return render(request, "<template>", {"objects": objects})
```

To minimize the amount of changes to our views file, we can just query the database and populate our list object. We will use Django queries to get our QuerySet. That way we can still pass in the dictionary object to the render.

One of the features that comes built into Django is admin functionality, located at localhost/admin when you start up the app. The admin functionality reads the metadata from your project, providing you with a quick interface for viewing and editing them. Its common practice to only give trusted users admin rights. An admin could log in, and see all the objects in the model, edit existing objects, add new objects, and remove existing objects.

Creating a Super User

```
> python manage.py createsuperuser
```

This command line command will create a super user who can use admin functionality in the project. Note that you will be prompted for a username, email, and password to use to log in as an admin.

Register Models with Admin

File: <project>/<app>/admin.py

```python
1 from django.contrib import admin
2 from .models import <model>
3
4 # Register your models here
5 admin.site.register(<model>)
```

Before an admin can interact with a model in localhost/admin, the model first needs to be registered, as they aren't be default.

We can do this by going into the app's admin file, and registering our model with the admin, as shown on line 5.