Django Outline

December 16, 2008

1 Introduction to Django

1.1 What is it?

Django is a web application **framework**, written in Python and designed to make writing web applications easier.

1.2 Why Django?

- It's fast: prebuilt components mean you can skip writing lots of the code you'd be writing in PHP
- It's easy: SQL is abstracted away so you don't need to worry about the database interaction layer[1]
- It's extensible: development is separated into different layers, which means that you can make large changes in your projet while maintaining sanity.
- No dollar signs: dollar signs suck

2 Prerequisite Knowledge

There's some stuff you should know (that you may not already know) before getting into Django.

2.1 Python

Django is a framework that's written on Python. As a result, you'll be doing most of your coding in Python. So, a bit of introduction about Python.

2.1.1 It's interpreted

Python is an interpreted language, which means, like PHP, no compiling. You can open up a Python interpreter by typing `python` at the command line. In this interpreter, you can directly type Python code and have it execute each line you type. *Note: the example code in this outline with lots of ">>>"s is what the code would look like typed into a Python interpreter.*

2.1.2 Whitespace

Python is cognizant of whitespace at the beginning of lines! Specifically, the indentation of lines indicates the block level of items. In C, braces ({ and }) determined block level; in Python, indentation does.

Warning: don't mix tabs and spaces in your code. To be safe, always use spaces (4 is a good number) instead of tabs, and make sure your text editors follow suit. If you like danger, just use the same text editor all the time.

[1] For our (and most) purposes anyway.

2.1.3 Variables

- are automatically declared the first time they are assigned
- have no type restrictions, just like in Javascript
- never need to be malloc'ed or free'd
- can only have local, global, or built-in scope

2.1.4 Some (Useful) Built-in Data Types

- Lists

  Like arrays in C, but actually useful.

```
>>> l = [False, 1, 2, "four", [4, 5]]
>>> l
[False, 1, 2, 'four', [4, 5]]
>>> l.reverse()
>>> l
[[4, 5], 'four', 2, 1, False]
>>> len(l)
5
>>> l[2]
2
>>> l[1:3]
['four', 2]
>>> l[:3]
[[4, 5], 'four', 2]
```

- Tuples

  Like lists, but immutable. Tuple packing and unpacking is useful for assignment

```
>>> t = (False, 1, 2, "three")
>>> t[2]
2
>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> #this is next line is crazy!!!!
>>> x, y = y, x
>>> x
2
>>> y
1
```

- Dictionaries

  Like the dictionaries you wrote, but way less ghetto.

```
>>> d = {'key': 'value', 1: 'value2', False: 7}
>>> d['key']
'value'
>>> d[1]
'value2'
>>> d[False]
7
```

2.1.5 For loops

All loops are over lists

```
>>> range(1, 4)
[1, 2, 3]
>>> for i in range(1, 4):
...     print 2*i
...
2
4
6
>>> l = ['a', 2, 'c']
>>> for j in l:
...     print j
...
a
2
c
```

2.1.6 Functions

Use the keyword `def`. Arguments can have default values (values given to arguments if they are not passed anything). Arguments can be passed by their keyword values.

```
>>> def divide_evens(x, y=2):
...     if x % 2 == 0 and y % 2 == 0:
...         return None
...     else:
...         return x / y
...
>>> divide_evens
<function divide_evens at 10042ed0>
>>> divide_evens(7, 4)
>>> divide_evens(8, 4)
2
>>> divide_evens(8)
4
>>> divide_evens(y=4, x=16)
4
```

2.2 Object Oriented Programming

Generalize structs into objects. Objects correspond to "things" in the real world. Objects can contain data and logic. Objects are defined by classes

```
>>> class Wallet:
...     """Stores money"""
...     def __init__(self, starting_money=0.0):
...         self.cash = starting_money
...
...     def spend(self, amount):
...         if amount > self.cash:
...             print "you're out of dough"
...             return None
...         else:
```

```
...             self.cash -= amount
...             return cash
...
>>> w = Wallet(30.0)
>>> w.cash
30.0
>>> w.spend(15.0)
15.0
>>> w1 = Wallet(50.0)
>>> w1.cash
50.0
>>> w.cash
15.0
>>> w1.spend(20.0)
30.0
>>> w.spend(20)
you're out of dough
```

Here, the code `Wallet(30.0)` creates a new Wallet object. Wallet objects can than use the functions inside the definition of Wallet. Every time you call a function from the an object, Python implicitly passes the object to the function. The following two function calls are equivalent (although the second one is the one you should be using):

```
>>> Wallet.spend(w, 1.0)
29.0
>>> w.spend(1.0)
28.0
```

Inheritance:

```
>>> class MagicWallet(Wallet):
...     """stores money, is magical"""
...     def __init__(self, starting_money=0.0):
...         self.start_cash = starting_money
...         self.cash = self.start_cash
...     def replenish(self):
...         self.cash = self.start_cash
...         return self.cash
...
>>> mw = MagicWallet(40.0)
>>> mw.cash
40.0
>>> mw.spend(30.0)
10.0
>>> mw.spend(20.0)
you're out of dough
>>> mw.replenish()
40.0
>>> mw.spend(20.0)
20.0
```

3 Djigg

We're going to recreat a something similar to digg.com. People submit links into some channels, other people can vote these links up or down, and the front page shows the links with the highest overall vote.

3.1 Installing Django

3.1.1 Windows

Go to Instant Django `http://instantdjango.com`. Follow the instructions. Easiest install of your life.

3.1.2 Mac / Linux / Unix

Download Django here: `http://www.djangoproject.com/download/`; follow the instructions under Option 1.

To make sure your install works, go into the command line, type `python`, and in the Python interpreter, type `import django`. Nothing should happen.

3.2 Starting our Project

1. `cd` to the folder where you want your project directory to be created.
2. Type

```
django-admin.py startproject djigg
```

3. Configure the database settingsSet the appropriate values for the DATABASE_* variables.

```
cd djigg
nano settings.py
```

4. Make sure that your settings are correct. Then, type:

```
python manage.py syncdb
```

5. ???
6. Profit

3.3 Creating an App

Django projects are divided into "apps" that encapsulate functionality. In many cases, it's ok to just have 1 app that does everything. However, if your site has a blog, a polls feature, and a digg-style feature, you should have 3 apps.

Let's create an app for our main feature. We'll call the app 'links'.

```
python manage.py startapp links
```

This should create a 'links' directory inside your djigg project.

3.4 Writing the models

Models represent your data as objects. In our example, we have links (that people submit), voting for those links, and channels. Put this in `links/models.py`.

```
from datetime import datetime
from django.db import models
class Channel(models.Model):
    name = models.CharField(blank=False, max_length=100)
    def __unicode__(self):
        return self.name
class Link(models.Model):
    title = models.CharField(max_length=100, blank=False)
```

```
    url = models.URLField()
    created_on = models.DateTimeField(default=datetime.now())
    votes = models.IntegerField(default=0)
    channels = models.ManyToManyField(Channel)
    def __unicode__(self):
        return self.title
```

The first two lines are like PHP includes. They load prerequisite libraries.

The class definition Link defines our link object. Links on djigg have a title, which is represented in the database as a VARCHAR datatype, a url, which is represented in Django as a regular string (but gets special processing by the URLField), a timestamp, for when the link was submitted, and a vote count. There's also a channels attribute. This relation to the Channel class is called a many-to-many relation, since each Link can be submitted to multiple channels, and each Channel can have multiple links. The __unicode__ function is what gets called when Django asks for a string representation of a *Link*. If Django (or you) want to turn a *Link* into a *String*, it'll call `__unicode__`.

Channel is pretty self explanatory.

3.4.1 The Arduous Process of Creating Tables in Your Database

You'd think that since I've already defined what my objects looked like, there would be some good way to get that information into the databse tables, without having to create the tables in something like PHPMyAdmin by hand. You'd be right. There's two steps.

1. In `settings.py`, add your app to the list of installed apps by inserting the line

   ```
   'djigg.links',
   ```

2. Run

   ```
   python manage.py syncdb
   ```

That's it. You probably won't ever have to look at your tables again.[2]

3.5 Admin Panel

Now, I'd starting writing code to let people upload links, but that would be kind of difficult without channels, right? How will we get channels into the database? Write SQL queries by hand? No. God no.

3.5.1 Change Your Settings

Head over to your `settings.py` file, and in value (it's a tuple) for the variable `INSTALLED_APPS`, add

```
'django.contrib.admin',
```

3.5.2 Change Your URL Configuration

Go to `urls.py` and add the lines[3]

```
from django.contrib import admin
admin.autodiscover()
```

Inside the value for the `urlpatterns` variable, uncomment the line

```
(r'^admin/(.*)', admin.site.root),
```

We'll keep this process mysterious for now, as url configurations will be covered later.

[2]The one caveat is that if you make changes to **existing** models, you have to either manually modify your tables, or clear your database and run syncdb again. In this department, Rails is a lot better.

[3]If you did everything right, the urls.py file should already have those two lines, except commented out. To uncomment them, just remove the # before the line.

3.5.3 Customize the Admin Panel

Create the file `links/admin.py`, and fill it with these lines:

```
from django.contrib import admin
from links.models import Comment, Link, Channel
admin.site.register(Link)
admin.site.register(Channel)
```

In `urls.py`, the line `admin.autodiscover()` instructed Django to look for files named `admin.py` inside each of your apps, and run the contents. In this admin definition file, we've simply registered the models we want in our admin panel.

The admin interface is actually highly customizable, but right now, its not really necessary.

3.5.4 Setup the Database (Again)

Since we've installed a new app, we need to run .

```
python manage.py syncdb
```

again, so that the tables for the admin app get generated. This time, it'll ask you to create a superuser. Do it.

3.5.5 Magic

Start the development server

```
python manage.py runserver
```

Point your web browser to `http://localhost:8000/admin`. Now we can add some channels or links

3.6 See the Links

Let's write the feature that allows people to see the links that people have added to the site (even though people can't currently do that).

3.6.1 Views

Views are the code that control what happen when someone visits some page. They are defined as functions[4], and are called by Django when people visit certain URLs (more on that later). Views generally load some data, and then pass that data off to a template to actually display the page.

Open up `links/view.py`. Let's write the view `index`, that will be responsible for displaying the homepage. Here, we just want to get all the most popular links from the database.

```
from django.shortcuts import render_to_response
from djigg.links.models import *
def index(request):
    links = Link.objects.all().order_by('-votes')[:10]
    return render_to_response('index.html', {'links': links})
```

Imports are at the top. We import all the models from the models definition inside the links app, since we'll need to use the models.

View functions take a request object as their first argument. This request object contains information about the user's request (such as GET and POST parameters).

The next line loads the first ten links with the most votes. Let's walk through this method call by method call:

[4] or any callable

- `Link.objects` : Link is a class, not an object. Django has given the Link class an objects attribute that helps us load objects from the database. Specifically, this call returns a *manager*.
- `.all()` : This is self-explanatory, this method returns all the Link objects in the database.[5]
- `.order_by('-votes')` : The Link objects returned are then ordered by the attribute votes, in descending order (ie, highest votes first).
- `[:10]` : This restricts the results to the first 10. If we wanted the last 10, we could use `[10:]`.

Last line actually returns HTML. The render_to_response function renders a template and returns a HTTP response object. Django sends whatever HTTP response is returned from the function back to the requester. render_to_response takes two arguments, the name of the template, and a dictionary of values[6] that will be exposed to the template.

3.6.2 Template

It's best to store templates inside apps. Create the folder `links/templates` and the file `links/templates/index.html`. render_to_response will automatically look for templates in folders called template inside app directories, so it'll find index.html . Let's write it:

```
<html>
<head>
    <title>Djigg!</title>
</head>
<body>
    <ul>
        {% for link in links %}
            <li>
                {{link.votes}} Votes: <a href="{{link.url}}">{{link.title}}</a>
            </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Notice that most of this is just regular HTML. Inside it, you'll see some tags delimited with curly braces and percent signs. There are two types of these Django tags.

1. **Template Tags**: These are begin with {% and end with %}. These usually execute some logic. Here, we hae a for loop, that loops through the items in the variable links. Remember that in the view, we exposed the variable links as 'links'.
2. **Variable Tags**: These are kind of like PHPs echo. They just dumpt the value of the variable into the template.

3.6.3 URL Configuration

Now we need to map URLs to views, so that when users visit urls, views will be called. We can define the url that will take the user to the home page. Open up `urls.py`.

There's one main thing going on here, and that's the definition of the variable `urlpatterns`. This is where all the url information is stored. When someone sends a request to the Django server, Django searches

[5] Ok, I lied. It doesn't actually load all of the objects from the database. These method calls construct a query, which only gets evaluated when absolutely necessary. The actual SQL statement generated would look something like:

```
SELECT * FROM links_link ORDER BY votes DESC LIMIT 0, 10
```

[6] which gets converted into a Context object

through the entries in `urlpatterns` for something that matches the url. That's why when we went to `/admin`, Django knew to take us to the admin application. Django uses regular expressions to match urls with entries. Regular expressions are an extremely powerful way to identify strings. They're not too hard to learn, but I won't go into detail.

Here's two additional lines that should go in the definition of `urlpatterns`:

```
(r'^$', 'djigg.links.views.index'),
(r'^home/$', 'djigg.links.views.index'),
```

The first entries redirects everyone visiting the root of your website to your index view. The first item in the tuple is a raw string (raw strings have an r before the first quote), which means that you don't have to escape any characters inside the string. The ^ means the beginning of the url (Django cuts off the beginning of the url; so if I requested `http://djigg.com/home/`, the url that would be matched would be `home/`), and the $ means the end of the url.[7]

3.6.4 Try It!

Browse on over to `http://localhost:8000/` or `http://localhost:8000/home/`, and you'll see the amazing frontpage!

3.7 Adding New Links

So we have an admin interface for adding new links, but we should also have a public interface for users to submit links.

3.7.1 Writing Forms

We need a form for users to upload links, but we shouldn't have to write these. Two reasons:

1. We already wrote all the information about what kind of fields exist in a link object. We shouldn't need to write this again!
2. Writing forms is boring.

In Django, forms are objects, just like Links or Channels. We just write up a description of what needs to go in the form, and Django handles rendering the form, validating the input, and converting form data into usable data. But since we've already written most of the attributes of the form already (in the Link class), we don't need to do that again.

Create the file `links/forms.py`. This is where we'll save forms we create.

```
from django.forms import ModelForm
from djigg.links.models import Link
class LinkForm(ModelForm):
    class Meta:
        model = Link
        exclude = ('votes', 'created_on')
```

The exclude attribute indicates which fields we want the form to exclude.

[7] We include the $, because regular expressions will match any string that fits the criteria, so if we had `r'^'`, every single url would be matched.

3.7.2 Working with Forms

Let's write a view to send the form to the user and to handle the input. Open up `links/views.py` and add the following:

```
from django.http import HttpResponseRedirect
from djigg.links.forms import LinkForm
from djigg.links.models import Link
def add_link(request):
    if request.method == 'POST':
        form = LinkForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/')
    else:
        form = LinkForm()
    return render_to_response('add_link.html', {'form': form})
```

Now the same view both displays the form and processes the form once the user submits it. The first part of the view checks to see if the user submitted something via POST. If so, the view attempts to save the response from the form, but only if the form is valid[8]. If the user does not submit something, the view creates a new LinkForm and renders it.

You probably noticed that this time, we use the template `add_link.html`, which doesn't exist yet. Go ahead and create `links/templates/add_link.html`.

```
<head>
    <title>Djigg! - Add a Link</title>
</head>
<body>
    <form action="." method="POST">
    {{ form.as_p }}
    <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

The form only generates fields for us, so that we can customize the form handler, method, and submit button(s).

Last thing we need is a url. Open up `urls.py` and add the entr y

```
(r'^add_link/$', 'djigg.links.views.add_link'),
```

Let's try it. Browse over to `http://localhost:8000/add_link/` and you'll see our form.

3.8 Voting

Now that people can add links, we want people to be able to vote. [9]

3.8.1 View

Open `links/views.py` and add the following

[8] Django automagically validates fields. For example, it validates that EmailFields only contain emails, and that IntegerFields only contain integers. You can also add custom validation in the form definition.

[9] The following is horrible design: there's nothing preventing someone from voting over 9000 times by visiting the link over and over. Additionally, POST requests should be made for requests that change the server

```
from django.shortcuts import render_to_response, get_object_or_404
def vote(request, link_id, updown):
    link = get_object_or_404(Link, pk=int(link_id))
    link.votes += updown
    link.save()
    return HttpResponseRedirect('/')
```

Notice that this view takes two arguments. `link_id` is the primary key of the link submission to vote for, and updown is either 1 or -1. At the end, the view redirects the user back to the home page.

3.8.2 URL

Add the following lines into `urls.py`

```
(r'^vote/(\d+)/up/$', 'djigg.links.views.vote', {'updown': 1}),
(r'^vote/(\d+)/down/$', 'djigg.links.views.vote', {'updown': -1}),
```

Inside regular expression, the parts that have parenthesis around them are "captured", and passed to the view function. In this case, the `(\d+)` matches and "captures" a positive number of digits (hence the \d) and sends them to the vote view. Additionally, the dictionary at the end of the url specification is a dictionary of additional arguments that get passed to the view function. Here, we pass the digits and either 1 or -1 to the vote view.

3.8.3 Template

Now we'll add these links to the home page, so people can vote for links there. Open up `links/templates/index.html` and modify it so the part in the for loop looks like this

```
<li>
{{link.votes}}
<a href="/vote/{{link.pk}}/up/">Up</a>,
<a href="/vote/{{link.pk}}/down/">Down</a> |
<a href="{{link.url}}">{{link.title}}</a>
</li>
```

3.9 Channels

Let's add the ability to view links by channels.

3.9.1 Views

Let's add a channel view. In `links/views.py`, add

```
def channel(request, ch_name):
    channel = get_object_or_404(Channel, name__iexact=ch_name)
    return render_to_response('channel.html',
        {'channel': channel, 'links': channel.link_set.all()})
```

`iexact` means case insensitive. Notice that we didn't define a `link_set` attribute on the Channel object, but since we defined a many_to_many relationship on the Link object, the Channel object knows has a link_set attribute that contains all the Link objects that are associated with the given Channel object.

3.9.2 Templates

Instead of creating a new template from scratch, we'll make a new template that inherits from the `index.html`, since that already defines a lot of the behavior we already need. First thing to do is change `links/templates/index.html`

```
<body>
    {% block before_links %}

    {% endblock %}
    <ul>
```

We've added a block in the index template. Now, other templates that extend index can override that block. Let's create `links/templates/channel.html`

```
{% extends "index.html" %}
{% block before_links %}
    <h3>{{channel.name}}</h3>
{% endblock %}
```

This template will now be the same as index, except that in index, where there's nothing in the block, we have the name of the channel.

3.9.3 URL

Let's create a url configuration for the channel view. Open up `urls.py` and insert the entry

```
(r'^channel/([A-Za-z]+)/$', 'djigg.links.views.channel'),
```

4 Advanceder Stuff

4.1 Static Media

Django is designed for serving up dynamic pages, not static pages, so it's probably best not to use Django for static stuff (like images, css, javascript, etc.). For sites that go on the Interwebz, you should use something separate. However, for development (and CS50 final projects), you can configure Django to serve static content. Use this guide found here: `http://docs.djangoproject.com/en/dev/howto/static-files/`

4.2 Files / Images

You want people to be able to upload files and images. Use the built-in file and image system! Put FileFields in forms or classes and then you can work with File or Image objects. For more details about files, take a look at: `http://docs.djangoproject.com/en/dev/topics/http/file-uploads` . Images are handled similarly.

4.3 Tags / Filters

The Django templating system is not very powerful[10]. But, it does have tags and filters that help you present your data. Filters get applied to variables, and they help "massage" your data. Filters get applied to variables with the `|` operator. Filters also return variables, so you can chain filters. For example, if we had the variable `str` with value `"Cansu is from Turkey?"`, we could write:

```
{{ str|lower|truncatewords:2 }}
```

[10]This is by design. It's good practice to keep separate the different layers of your application. Django sort of forces you to do that with the template layer.

and it would show up as

```
cansu is ...
```

Beside `block` and `for`, there are also a bunch of tags that come with Django. For the full list, look here: `http://docs.djangoproject.com/en/dev/ref/templates/builtins/`

4.4 Authentication

Django has a built-in authentication framework. If the `django.contrib.auth` app is installed (in your INSTALLED_APPS setting), you get the `User` model[11]. You can create users, let users login, etc. Then, in your views (that need authentication), include the code[12]

```
if not request.user.is_authenticated():
    retun render_to_response('myapp/error/or/something/')
```

There's also built-in views for loginning and logging out. For all the details, take a look at `http://docs.djangoproject.com/en/dev/topics/auth/`.

4.5 Sending Emails

Django has helpers for sending emails. They're located in `django.core.mail`. The mail function is `send_mail()`, which does exactly what you think it does. You call it, and an email gets sent. You can also use template rendering to render HTML (or even just plain text) from your a template, and then send that HTML with `send_mail`. For the full documentation, look here: `http://docs.djangoproject.com/en/dev/topics/email/`.

4.6 Deployment

For development, `python manage.py runserver` works well, but if/when you actually want your webapp to show up on the webernetz, you'll probably want to deploy on Apache with mod python. This isn't too bad if you have your own web server. You can folllow the directions here: `http://docs.djangoproject.com/en/dev/howto/deployment/modpython/` No instructions just yet for cloud.cs50.net, but if people are interested, I'll to post some instructions at some point.

4.7 Customizing the Admin

The admin interface is amazingly powerful, since it generates so much functionality with so little code. The default probably admin site probably won't be exactly what you want though. Fortunately, there are tons of ways to customize the automatically generated admin app. You can choose which fields you want to include / disinclude, can add filters to pages, and control how items are ordered, among many others. Most of these are defined in the `admin.py` files, and all the specific options can be found here: `http://docs.djangoproject.com/en/dev/ref/contrib/admin/`.

4.8 Middleware

Middleware hooks into Django's request and response processes. Django has a bunch of built-in middleware that does things like manage sessions and inserting headers into responses. You can enable / disable these in your settings file. You can, however, write your own. You can use middleware to do some sort of processing to all requests / responses. You could, for example, use middleware to take the user's IP address, convert it ito a zipcode, and append it to the request, which you can then process in your views.

[11] You do, however, have to do `fromdjango.contrib.auth.modelsimportUser`

[12] If you're familiar with Python, you can also use the decorator `login_required()`

5 Appendix

- The Django Website: `http://djangoproject.com/`. There, you can find downloads, documentation, and tutorials!
- The Python Website: `http://docs.python.org/`. Same description as for the Django website!
- The Django Google Group: `http://groups.google.com/group/django-users`. If you have questions, they usually respond to emails pretty quickly (but read the documentation before you send emails to them).