Jeff Croft
October 16th, 2008
The Web Design Conference, Orlando

Introduction to Django

Introduction

I’m Jeff Croft.

What do I do?

- Design and develop web and mobile sites and applications for Blue Flavor and its clients.
- Use and evangelize Django, an open source, Python-based web application framework that originated at The World Company, my former employer.
- Write about web design, Django, Mac nerdery, and whatever else strikes my fancy at jeffcroft.com.

What is Django?

From the horse's mouth:

"Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design."

djangoproject.com

“...high level Python web framework...”

- A web framework attempts to abstract routine development tasks by providing shortcuts.
- For example: dealing with the details of HTTP, connecting to a database, managing users and groups, generating feeds, etc.
- However, a good web framework shouldn’t get in your way if you want or need to work at this lower level. You should always be able to “step down a level.”
- Django is built on Python, an established high-level language used heavily at Google, Yahoo, CERN, and NASA.
- Django makes a point of not changing anything about the way Python works. If you learn Django, you’re learning Python—and vice-versa.

“...encourages rapid development...”

- Saving you time is one of Django’s primary goals.
- Django was created in the fast-paced news world, and was designed around a real-world need to develop sites in a matter of hours, not days or weeks.

“...clean, pragmatic design.”

- Django is opinionated software. It firmly believes there is a “right” way to do many things.
- It tries to make it as easy as possible to do things the “right” way. Django helps you subscribe to best practices by making them simple.

A bit of Django history

- Created in 2003 at The World Company in Lawrence, KS, by Adrian Holovaty and Simon Willison. Originally, it was call “The CMS”, and was a project to build a CMS for newspapers.

- As time passed, they abstracted the underlying “stuff to help you build web apps” from the higher level “stuff that makes running an online newspaper easier.” The two became Django (open source) and Ellington (commerical), respectively.

- Django was released as open source software in the summer of 2005. Since then, it has really taken off, and is in use at many major corporations, include Google.

- Predictably, Django became extremely popular in the journalism world. In 2007, there were 14 Django-powered sites among the finalists in 11 categories for the Digital Edge online journalism awards. Most of these were running Ellington.

Continued...

- The core Django developers had several big changes they wanted to make that would introduce backwards-incompatibilities for developers who'd built sites on Django. They decided to make these changes before giving Django the 1.0 version number. As such, the time between Django's original release and it's 1.0 release was...well, ridiculous.
- Django 1.0 was released on September 2nd, 2008!
- The good news for you: there should be no significant backwards incompatible changes from here on out. The code you will learn today will be valid Django code for a very long time.

Projects and apps

Projects in Django

- A project is a configuration for an instance of Django.
- It contains things like database settings, filesystem paths, and so forth. It may also contain settings for applications (more on applications soon).
- It also contains a list of the applications your project will be using.
- Usually, a project is associated with a single website—each website has a project that controls it. There are cases where a single project can drive multiple websites, but that's more advanced and not for you to worry about right now.

Apps in Django

- An “app” in Django is a collection of code that makes up some sort of functionality for your website.
- A typical project will use several apps.
- Apps can be reused between projects (this is one of Django’s coolest features — more on it later).
- Deciding what is an “app” can be tricky. For example, should you make an app that handles your blog and comments, or should you make two apps (one for blog, one for comments)?
- Generally, it’s best to make two apps. But don’t worry about this too much right now. You can always refactor your code into multiple apps later, when you start to see how it might be beneficial.

Apps are “pluggable”

- One of Django’s greatest strengths is that a single app can be reused in many projects.
- This has fostered a great community around building reusable apps that are made available for you to just “plug in” to your project.

Let’s say you have three sites:

- First, you have a personal site. It includes a blog with tags and comments, a photo gallery, some “static” pages, and contact form.

- You also have a Digg-style social news site. It contains stories with tags and comments, the ability for users to vote on stories, some “static” pages, and a contact form.

- You also have a movie review site. It contains a database of movies with tags, your reviews with tags and comments, the ability for users to vote on movies, some “static” pages, and a contact form.

There's some overlap here.

- All three sites have comments, "static" pages, and a contact form.
- Two sites have blogs and photo galleries. Two sites also allow users to vote on items.
- Ideally, you build generic apps for these functions, so that they can work across all three sites. Clearly, this is better for maintainability.

These already exist.

- Django comes with a comments app. It allows users to comment on any sort of item in your database (blog entry, story, movie review, etc.)

- Django also comes with an app for handling “static” pages (simple pages with content stored in the database).

- django-tagging (http://code.google.com/p/django-tagging/) allows you to easily add tag functionality to any sort of item in your database.

- django-voting (http://code.google.com/p/django-voting/) allows users to vote on any sort of item in your database.

- django-contact-form (http://code.google.com/p/django-contact-form/) provides basic contact form functionality.

- There are several available blog and photo gallery applications.

django.contrib

- Django is distributed with several of these reusable apps. They're all optional, but available for use in your project if you want them. Some of them are:

- django.contrib.admin — Provides an automatic admin interface
- django.contrib.auth — Provides users and groups
- django.contrib.comments — Provides commenting functionality
- django.contrib.flatpages — Handles simple, "static" pages
- django.contrib.syndication — Makes creating feeds easy

More examples

- All of the following are available on Google Code (as well as a ton of others — these are just a few I thought were useful):

- django-pagination
- django-mailfriend
- django-messages
- django-forum
- django-profiles
- django-friends
- django-registration
- django-authopenid
- django-basic-apps

Installing an app in a project

- In order to use your app in a project, you need to place a reference to it in the INSTALLED_APPS list in your settings.py file.

```
INSTALLED_APPS = (
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.sites",
    "registration",
    "pagination",
)
```

Models

What are models?

- A model is a description of a particular type of data in your application.
- Your models define what types of data will be available in your application and what fields and attributes those objects have.
- In Django, model definitions also contain metadata about your models.
- In Django, models map one-to-one to database tables.

Typically, you'd do this:

```
CREATE TABLE "people_person" (
    "id" integer NOT NULL PRIMARY KEY,
    "first_name" varchar(300) NOT NULL,
    "last_name" varchar(300) NOT NULL,
    "bio" text NOT NULL,
);
```

But not in Django.

- SQL is not consistent from database to database. Django projects should easily port from one database environment to another without having to change your code. Django models work with any database Django supports.

- If you store your models in the database, it becomes difficult to keep them under version control.

- Your application will need to work with the model regardless of how you created it. If you created it in SQL, you'd need to recreate it in Python so your application can work with it. Thus, Django just has you create models in Python from the start, and handles the SQL for you.

- Writing Python is fun! Well, at least it's a lot more fun than writing SQL. If you ask me, anyway. :)

The same model, in Django:

```
from django.db import models

class Person(models.Model):
    first_name  = models.CharField(max_length=300)
    last_name   = models.CharField(max_length=300)
    bio         = models.TextField()

    def __unicode__(self):
        return (%s %s) % (self.first_name, self.last_name)

    class Meta:
        ordering = ["last_name"]
        verbose_name = "Person"
        verbose_name_plural = "People"
```

The same model, in Django:

```
from django.db import models

class Person(models.Model):
    first_name  = models.CharField(max_length=300)
    last_name   = models.CharField(max_length=300)
    bio         = models.TextField()

    def __unicode__(self):
        return (%s %s) % (self.first_name, self.last_name)

    class Meta:
        ordering = ["last_name"]
        verbose_name = "Person"
        verbose_name_plural = "People"
```

Models can contain relationships

```
from django.db import models

class Person(models.Model):
    first_name  = models.CharField(max_length=300)
    last_name   = models.CharField(max_length=300)
    bio         = models.TextField()
    ...

class Article(models.Model):
    title       = models.CharField(max_length=300)
    date        = models.DateTimeField()
    author      = models.ForeignKey(Person,related_name= "articles")
```

Syncing your database

- Once you've created models, Django will run the necessary SQL to install them into the database for you. You just need to run the manage.py command "syncdb"

```
$ ./manage.py syncdb
Creating table my_first_model
Creating table my_second_model
...
```

- This will install all models for any app in your project's INSTALLED_APPS setting.

Django's database API

- Once you've installed your models, Django provides a high-level API for working with them.
- The API provides functions for creating, retrieving, changing, and deleting objects in the database — and a lot more.

Creating an object

- Assuming the “Person” model I showed earlier, you can create a new person in the database as follows:

```
p = Person(
    first_name = "Jeff",
    last_name = "Croft",
    bio = "Jeff Croft is awesome."
)
p.save()
```

- Note the explicit .save() method each model gets. This is an important bit of Django philosophy: Django never hits the database until you explicitly tell it to. Thus, the .save() call is required.

Retrieving objects

- Each model gets a member called “objects,” which is used for data retrieval. A few examples:

```
Person.objects.all()
Person.objects.get(id=10)
Person.objects.filter(first_name="jeff")
Person.objects.exclude(first_name__contains="je")
Person.objects.all().order_by("first_name")
Article.objects.filter(author__first_name="jeff")
```

- You can chain these things. You can do, for example: `Person.objects.filter().exclude().order_by()`

- The Django database API is very powerful, and I’ve personally never come across a situation where it couldn’t do what I needed it to do. But, if there is a case when you need to write SQL directly, Django will let you do that.

Using relationships

```
p = Person.objects.get(first_name="Jeff", last_name="Croft")
p.articles.all()
p.articles.filter(title__contains="django")

a = Article.objects.get(title="Django is great")
a.author
a.author.first_name
```

Editing an object

```
p = Person.objects.get(first_name="Jeff", last_name="Croft")
p.bio = "No, really. Jeff Croft is totally awesome."
p.save()
```

Deleting an object

```
p = Person.objects.get(first_name="Jeff", last_name="Croft")
p.delete()
```

Lazy QuerySets

- When you pull items from the database, Django calls the result a “QuerySet.” It’s more or less a Python list with some added functionality. QuerySets are lazy. That is to say, they only hit the database when they're evaluated. For example:

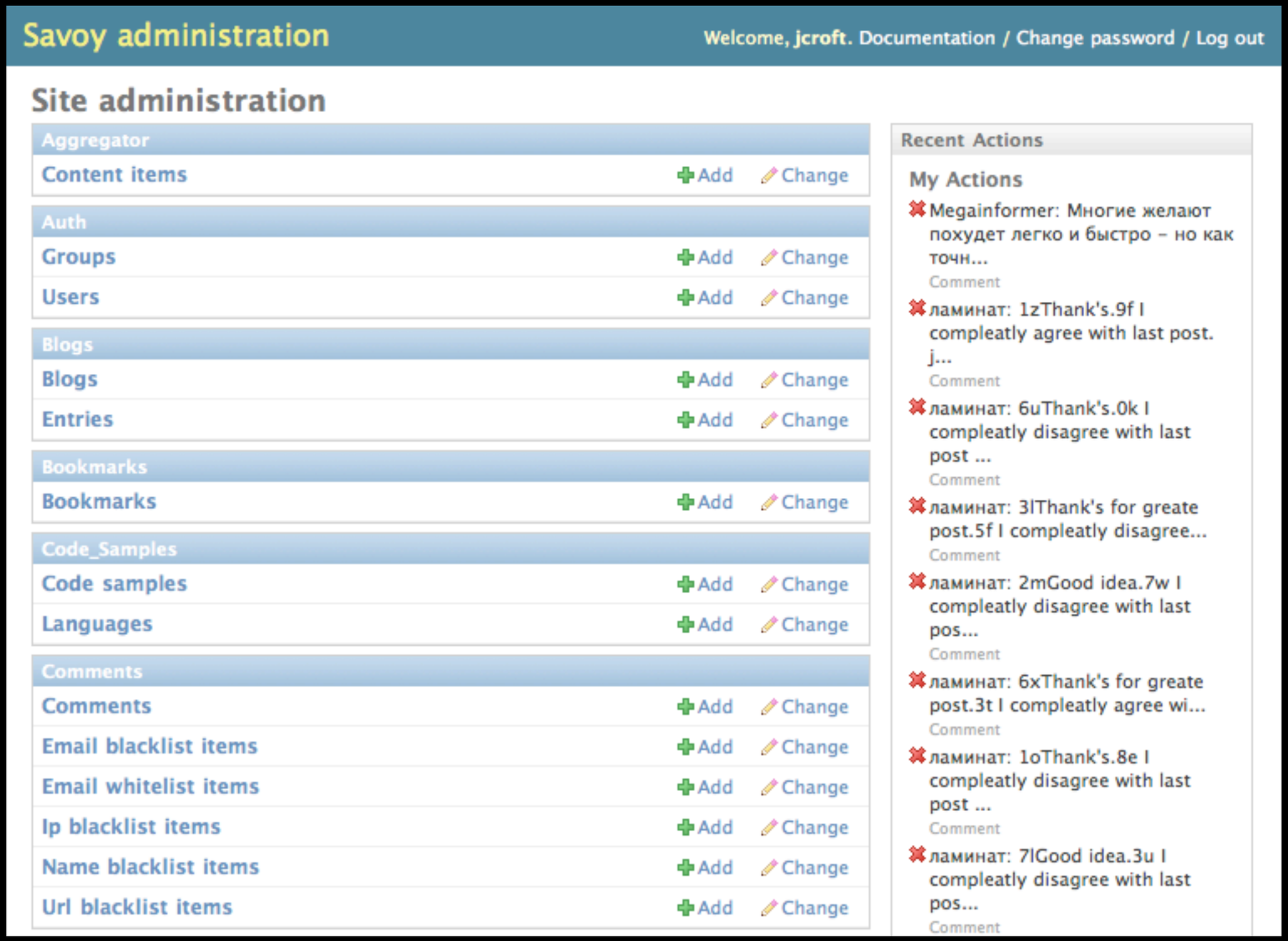
```
people = Person.objects.filter(first_name="jeff")
people = people.exclude(last_name__contains="croft")
people = people.order_by("last_name")
print people   # Database hit
```

- This code executes exactly one database query.

Django admin

What is the Django admin?

- The Django admin is one of the afore mentioned optional "contrib" apps that are distributed with Django itself.
- It provides a powerful, customizable, production-ready interface to creating, updating, and deleting instances of your models.
- It includes data entry validation, user and group permissions, and a log of user activity in the Admin.
- It is designed for regular users. It's not like phpMyAdmin or other database administration tools. It was built at a newspaper so that editors and producers could enter and update content.
- The Django admin works off your models alone. Even if your app is not done, once you have the models in place, you can start working with the Django admin.


Savoy administration
Welcome, jcroft. Documentation / Change password / Log out
Site administration
Aggregator
Content items Add Change
Auth
Groups Add Change
Users Add Change
Blogs
Blogs Add Change
Entries Add Change
Bookmarks
Bookmarks Add Change
Code_Samples
Code samples Add Change
Languages Add Change
Comments
Comments Add Change
Email blacklist items Add Change
Email whitelist items Add Change
Ip blacklist items Add Change
Name blacklist items Add Change
Url blacklist items Add Change
Recent Actions
My Actions
Megainformer: Многие желают похудет легко и быстро – но как точн...
Comment
ламинат: 1zThank's.9f I compleatly agree with last post. j...
Comment
ламинат: 6uThank's.0k I compleatly disagree with last post ...
Comment
ламинат: 3lThank's for greate post.5f I compleatly disagree...
Comment
ламинат: 2mGood idea.7w I compleatly disagree with last pos...
Comment
ламинат: 6xThank's for greate post.3t I compleatly agree wi...
Comment
ламинат: 1oThank's.8e I compleatly disagree with last post ...
Comment
ламинат: 7lGood idea.3u I compleatly disagree with last pos...
Comment

Savoy administration

Welcome, jcroft. Documentation / Change password / Log out

Home › Entries

Select entry to change

Add entry

Go

2003 2004 2005 2006 2007 2008

Entry title	Date published	Author	Entry status	Enable comments
New: Podcast "conversations"	Aug. 12, 2008, 5:25 p.m.	Jeff Croft	Published	✓
"Default" templates in Django	Aug. 5, 2008, 12:07 p.m.	Jeff Croft	Published	✓
Top ten things that suck about Django, revisited	July 25, 2008, 11:34 a.m.	Jeff Croft	Published	✓
Fall speaking schedule	July 24, 2008, 10:05 a.m.	Jeff Croft	Published	✓
MobileMe doesn't support calendar subscriptions	July 11, 2008, 12:40 p.m.	Jeff Croft	Published	✓
Goin' (back) to Kansas City	June 30, 2008, 7:13 a.m.	Jeff Croft	Published	⊖
Why we don't skip Photoshop	June 4, 2008, 5:20 p.m.	Jeff Croft	Published	⊖
Boagworld interview	May 7, 2008, 8:51 p.m.	Jeff Croft	Published	✓

Filter

By date published
Any date
Today
Past 7 days
This month
This year

By Entry status
All
Draft
Published
Hidden

Savoy administration

Welcome, jcroft. Documentation / Change password / Log out

Home › Entries › Add entry

Add entry

Meta:

Author: ---------

Select the author of this entry.

Blogs: jcroft's blog

Select the blogs this entry belongs to. Hold down "Control", or "Command" on a Mac, to select more than one.

Posted from: ---------

Select the place from which this entry was posted.

Entry title:

Enter the title of this blog entry.

Slug:

The slug is a URL-friendly version of the title. It is auto-populated.

Date published:

Date: 2008-08-24 Today |

Time: 15:12:56 Now |

Select the date and time this entry was posted.

Why? And what if I don't want it?

- Writing admin interfaces sucks. It's boring, repetitive, and just usually not that much fun.
- The admin interface is entirely optional. It's just a Django app you plug into your project like any other. If you don't want it, you just don't plug it in.
- If you want or need to write your own administrative interface for your project, you can certainly do so.

Views

What are views?

- A view is (generally) a “type” of web page within your application.
- A view (generally) performs a specific function and renders a specific template.
- A view is simply a Python function that takes an HTTP request and returns an HTTP response.
- View functions can live anywhere in your Python path, but are typically put into the views.py file for your application.

A view is not “how it looks.”

- In most cases, the view is responsible for pulling together all the information that’s available at a given URL. It may look up information in the database, perform some kind of processing on data, process input from a form, send an e-mail, etc.

- Since a view is just a Python function, it can really do anything you want it to. The only rules are that it must take an HTTP request and return an HTTP response.

- The view shouldn’t be outputting (X)HTML directly. Rather, it will render through a template, which is where the (X)HTML is structured. We’ll deal with templates later.

A simple first view:

```
from django.http import HttpResponse

def homepage(request):
    return HttpResponse("Welcome to the homepage!")
```

A less simple view:

```
from django.shortcuts import render_to_response
from myproject.myapp import Person

def people_list(request):
    people = Person.objects.all()
    return render_to_response(
        "people/list.html",
        {"people_list": people},
    )
```

Another example:

```
from django.shortcuts import render_to_response, get_object_or_404
from myproject.myapp import Person

def person_detail(request, lastname):
    person = get_object_or_404(Person, last_name=lastname)
    return render_to_response{
    "people/detail.html",
    {"person": person},
}
```

Generic views

What are generic views?

- Generic views are a set of regular Django views that are distributed with Django and cover a large variety of standard use cases.

- They save you time, by making you not have to write views in “typical” situations. You can just use the included generic views, instead.

- In many cases, generic views are all (or most of what) you need. Simple sites often use exclusively generic views. At the newspapers I worked for, generic views covered around 85% of the pages on the site.

Django provides:

- django.views.generic.simple
- django.views.generic.date_based
- django.views.generic.list_detail
- django.views.generic.create_update

URLconfs

What are URLconfs?

- In Django, you explicitly define which URLs map to which functions. The URLs are not implicit from the filesystem, from the view name, or from anything else. You must consciously think about what your URLs should look like and how they should work.

- A “URLconf”, as Django people call them, is a mapping of URLs to view functions. For example, if someone visiting the URL http://yoursite.com, the URL conf points them to the view function you’ve written for your homepage. If they go to http://yoursite.com/blog, the URL conf points them to the view function you’ve written for your blog index. And so forth.

- Django uses regular expressions to match URLs.

- URLconfs can live anywhere in your Python path, but typically each application will have a urls.py file outlining the URLs it provides, and each project will also have a urls.py file.

Why? Because these are ugly.

- page.php
- script.cgi?pageid=144
- StoryPage.aspx
- 0,2097,1-1-30-72-407-4752,00.html
- /stories/147/

- URLs are part of your application design. You should think about them as a key part of the interface to your site and design them with the same level of attention you'd design any other interface element.

Django URLs look more like this:

- /stories/
- /stories/2008/apr/20/michelles-birthday/
- /categories/birthdays/

- Really, Django URLs can look any way you want them to—even the ugly ways in the previous slide. But, this sort of scheme is typical.

Templates

What are templates?

- If a view describes what data is on a page, then a template defines how that data is presented as HTML (usually).
- While Django's template language is most commonly used to generate (X)HTML, it can actually be used to create any text-based file format (e-mail, RSS, CSV, XML, YAML, etc.).
- Django's template language was designed to be usable by designers and HTML authors. It should not require a programmer to operate it.

An example template

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>People ({{ people_list|length }} total)</h1>
    <ul>
      {% for p in person_list %}
        <li>
          <a href="{{ p.last_name }}/">
            {{ p.first_name }} {{ p.last_name }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Three main template components

- **Variables:** Remember how our view provided a “context” to the template? That context is the set of variables our template has access to. Variables are used like this: {{ variable_name }}

- **Tags:** Tags are more complex than variables. Some create text in the template, some perform loops or logic, some add data to the current template context, and some do other stuff. A tag can do virtually anything. Tags may take arguments. Tags are used like this: {% tag_name %}

- **Filters:** Filters can be thought of like a pipe, in UNIX. You pass something into them, and they return something different. They can be strung together, too. Many filters reformat a variable. For example, the “date” filter defines how a date variable will be outputted. Filters are used like this {{ variable|filter_name }}

Variables

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>{{ person.first_name }} {{ person.last_name }}</h1>
    <p>{{ person.bio }}</p>
  </body>
</html>
```

Variables

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>{{ person.first_name }} {{ person.last_name }}</h1>
    <p>{{ person.bio }}</p>
  </body>
</html>
```

Filters

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>People ({{ people_list|length }} total)</h1>
    <ul>
      {% for p in person_list %}
        <li>
          <a href="{{ p.last_name }}/">
            {{ p.first_name|capfirst }}{{ p.last_name|striptags }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Filters

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>People ({{ people_list|length }} total)</h1>
    <ul>
      {% for p in person_list %}
        <li>
          <a href="{{ p.last_name }}/">
            {{ p.first_name|capfirst }}{{ p.last_name|striptags }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Tags

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>People ({{ people_list|length }} total)</h1>
    <ul>
      {% for p in person_list %}
        <li>
          <a href="{{ p.last_name }}/">
            {{ p.first_name }}{{ p.last_name }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Tags

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>People</title></head>
  <body>
    <h1>People ({{ people_list|length }} total)</h1>
    <ul>
      {% for p in person_list %}
        <li>
          <a href="{{ p.last_name }}/">
            {{ p.first_name }}{{ p.last_name }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Template inheritance

- Django's template language includes an inheritance mechanism that is both extremely powerful and sort of hard to get your head around at first.

- You might think of it as an updated (if backwards) take on "includes," which you may have used in the past (SSI, PHP includes, etc.)

- The core problem inheritance is trying to solve is one of repeated code. Ideally, you Don't Repeat Yourself (DRY), and any time you need to make changes to template code, you only have to do it in one place.

“Blocks”

- Django’s template inheritance centers around the idea of “blocks”. A template can define “blocks”—essentially chunks another template can override.

- A typical setup is to have a “base” template which provides a bunch of empty blocks, and then “child” templates which fill them. In Django parlance, child templates “extend” other templates.

- Extension doesn’t have to be only two level: you could have a “base” template for your site, plus a template for a section of your site which extends the base, plus a template that extends the section template. And so on.

A “base” template base.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>{% block title %}{% endblock %}</title></head>
  <body>
    <div id=”content”>
      {% block content %}
        <h1>Welcome to our site!</h1>
      {% endblock %}
    </div>
    <div id=”footer”>
      {% block footer %}
        <p>Copyright 2008 Jeff Croft</p>
      {% endblock %}
    </div>
  </body>
</html>
```

“Blocks”

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
  <html lang="en">
  <head><title>{% block title %}{% endblock %}</title></head>
  <body>
    <div id=”content”>
      {% block content %}
        <h1>Welcome to our site!</h1>
      {% endblock %}
    </div>
    <div id=”footer”>
      {% block footer %}
        <p>Copyright 2008 Jeff Croft</p>
      {% endblock %}
    </div>
  </body>
</html>
```

A child template person.html

```
{% extends “base.html” %}

{% block title %}
  {{ person.first_name }}{{ person.last_name }}
{% endblock %}

{% block content %}
  <h1>About {{ person.first_name }}{{ person.last_name }}</h1>
{% endblock %}
```

A child template person.html

```
{% extends “base.html” %}

{% block title %}
  {{ person.first_name }}{{ person.last_name }}
{% endblock %}

{% block content %}
  <h1>About {{ person.first_name }}{{ person.last_name }}</h1>
{% endblock %}
```

The beauty of inheritance

- Besides keeping everything DRY, one of the biggest reasons to make good use of inheritance is for redesigns: often, replacing the base.html template is all you have to do to get a totally new layout, look, and feel.

A note on tags and filters

- Django is distributed with a vast library of template tags and filters.
- As you get more advanced, though, you may wish for a template tag that doesn't exist.
- Template tags and filters are just Python functions, so there's nothing to stop you from writing your own. In fact, most Django apps provide some template tags (along with their models, views, and urls).
- To load a non-built-in set of tags (such as those an application might provide), use the "load" tag: {% load blog_tags %}

What else is in Django?

Some of the things we didn't cover:

- **Forms:** Handles HTML form display, data processing (validation) and redisplay.
- **Caching:** Flexible caching at any level, using memcached, database, local memory, or filesystem caching.
- **Internationalization:** Full support for internationalization of text.
- **Sessions:** Full support for anonymous, cookie-based sessions.
- **Testing:** Full testing suite included.
- **Middleware:** a framework of hooks into Django's request/response processing. Allow for global altering of Django's input and/or output.

Okay, go write some code!

(or ask me questions.)

Thanks, yo.