django

A (fast-paced) introduction

Jacob Kaplan-Moss

Colorado Software Summit
October 19th - 24th, 2008

http://toys.jacobian.org/presentations/2008/softwaresummit/tutorial/
Welcome!

My name is Jacob Kaplan-Moss; I’m one of the lead developers of Django and the former lead developer at the Lawrence Journal-World, where Django originated.

I’m going to do my best in the next ninety minutes to give you a solid introduction to Django. It’s an amazingly powerful framework, so we obviously won’t be able to touch on everything, but hopefully by the end we’ll have covered enough of the basics to get you a taste of what Django development “feels like.”

These are the basic points we’ll cover.

I’ll give a quick introduction to some of the philosophies behind Django, and talk a little bit about how to get Django up and running.

I won’t really spend a whole lot of time on either of those topics, though both are quite well documented on the website.

So the bulk of this tutorial will be spent on what I like to think of as the three main legs that Django stands on: Models, Views, and Templates (“MTV”). In other words, we’ll talk about how you define data, how you read and write data, and how you present that data to users.

Along the way we’ll cover one of the coolest advanced features of Django: the automatic admin interface.

Again, this is a quick introduction to a few bits of philosophy.
Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.

This is the one sentence introduction to Django; the “mission statement” as it were. Let me break it down.

A Web framework is software that abstracts common problems of Web development and provides shortcuts for frequent programming tasks.

A good web framework finds the “pain points” of web developers and smoothes them over — but never gets in the way! It should let you work at a much higher level of abstraction, so you don’t need to worry about details of HTTP, SQL, or whatever. Again, it shouldn’t get in your way if you need to “step down” a level.

Python itself is another key “feature” of Django. It’s a beautiful, concise, powerful, high-level language with an amazing community; many things are easier in Django simply because of the tools you can build on top of.

Django also makes a point of not changing anything about how Python works; if you learn Django, you’re also learning Python. This helps down the road.
Regardless of how many powerful features it has, a Web framework is worthless if it doesn't save you time. Django's philosophy is to do all it can to facilitate hyper-fast development. With Django, you build Web sites in a matter of hours, not days; weeks, not years.

This comes directly out of real-world problems. We’re programmers, yes, but we work at a news organization. When a big story breaks, we don’t have the luxury of a long development cycle. Every convenience in Django is there because it makes you more productive.

Django strictly maintains a clean design throughout its own code and makes it easy to follow best Web-development practices in the applications you create.

The philosophy here is to make it easy to do things the "right" way.

More Django philosophy:

» http://www.djangoproject.com/documentation/design_philosophies/

» http://www.djangobook.com/en/1.0/chapter01/
One day, someone will finally win the OS wars and I’ll be able to quickly walk everyone through installing Django. Until then, the documentation linked here is pretty damn good.

» http://www.djangoproject.com/documentation/install/

» http://www.djangobook.com/en/1.0/chapter02/

» http://code.djangoproject.com/wiki/SetupOnTiger

» http://code.djangoproject.com/wiki/WindowsInstall

I actually think Django’s pretty easy to install — you only need Python and a database — but multiplied by a thousand different environments it gets hard to “teach.” Now that Python2.5 includes SQLite, it’s gotten a lot easier, so if you’re having problems I suggest starting there.

If you’ve got a working install, however, you should be able to follow along with the rest of the talk pretty well...

Note that to follow along properly you should install Django from SVN; this tutorial uses features found only in the development version.

The first time you run Django, you need to create a “project”. A project is a collection of settings for an instance of Django — including database configuration, Django-specific options and application-specific settings. You can think of a “project” as a single website (though one project could drive multiple sites).
Obviously we need a project for the tutorial.

Our project:

Here’s what we’ll be building. The essential idea is “cheeseshop + digg”; a site where people can give a thumbs-up or -down to packages registered with the Python Cheeseshop. We’ll import data using the ‘shop’s XML-RPC interface, mash it up with votes, and put a somewhat pretty face on it.


So, we have to start by creating a project. This command creates a project named “cheeserater” in the current directory.

```bash
$ django-admin.py startproject cheeserater
```

And here’s what you get from the created project.
$ ./manage.py runserver
Validating models...
0 errors found.

Django version 1.8, using settings 'cheesereater.settings'
Development server is running at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.

Django comes with an internal development server. It’s a lightweight, pure-Python Web server that builds on the HTTP server included in Python's standard library. We've included this with Django so you can develop things rapidly, without having to deal with configuring Apache until you're ready for production. This development server watches your code for changes and automatically reloads, helping you make many rapid changes to your project without needing to restart anything.

It’s also a great way to make sure everything’s set up correctly.

It worked!

The next step is to edit settings.py with your project settings. There’s quite a few options there (see http://www.djangoproject.com/documentation/settings/ for all of ‘em), but the DATABASE_* ones are the most important ones at first.

Keep in mind that that settings.py is just a normal Python module with variables, so you can include if statements, import statements, whatever.
Run `syncdb` to synchronize your database with what’s in your settings files. You’ll run this the first time you get a Django project up and running, and then again every time you add models to your project (more on that later).

Django notices you’ve got the authorization system installed, and creates a superuser account for you. That’ll come in handy later when we play with the admin interface.

For more one settings and the `django-admin` utility, see:


Now that we’ve got the skeleton of a project working, we’ll start on an app. An app is just a collection of data and code that makes up some sort of logical “unit”. Typically, projects are composed of a number of apps; the cool part is that apps can be reused in multiple projects.

Typically, an app exists to solve a single problem.

Often, design of apps is iterative: you start with a single app with a bunch of code and over time refactor it into a set of smaller apps as the difference between pieces becomes more apparent. This isn’t exactly possible in just three hours, so I’ll try to summarize how I came to the design we’ll be using.
So these are the models we’ve got to deal with. I’ll talk more about models in just a little bit, but for now you can think of each of these as a database table.

So we’ll end up with three apps: one for the cheeseshop packages, one for votes, and one for users.

For users, we can use the `django.contrib.auth` package that ships with Django; we’ll augment it with `django-registration` ([http://django-registration.googlecode.com](http://django-registration.googlecode.com)).

Why break votes out into a different package? Well, mostly because it’s a somewhat separate concept from packages: the idea of “voting” has nothing to do with the idea of tracking Python add-ons. Case in point: originally voting was a part of cheeserater (`cheeserater.votes`), but it’s since been spun off into a generic package that provides voting on any model. So I’ve ditched the cheeserater-specific voting module in favor of the more generic `django-voting` ([http://django-voting.googlecode.com/](http://django-voting.googlecode.com/)) package.
Although relational databases were first conceived of in the 1960s and have been around in some form or another for many years, the web has really taken database use to a whole new level. Almost every website is backed by one database or another.

When done right, database-backed websites are extremely useful. Instead of the top-down style that dominates in print, databased websites let readers "choose their own adventures" as they move through the site.

The best sites take this to an extreme by providing tools that let readers sift through mountains of data they'd otherwise be unable to process. Helping developers build those tools is the primary purpose of Django. It makes sense, then, to start our discussion of Django development by focusing on how Django handles data modeling.

In a nutshell, a model describes your data. They're like a blueprint: they explain what types of objects you'll use, what fields those objects have, and how it all fits together. In Django, models definitions also contain metadata about your models that Django uses to build APIs -- more on that later.

This is the usual way of defining a model in a relational database, but Django doesn't do that.
Many web development platforms want you to define your models using SQL. After all, the data will eventually be stored in a database, so why not simply define your models there first? While this logic makes sense on the surface, we disagree:

» Writing SQL can be tough. Different databases require you to do things differently, and having to write three different versions of your data definitions is the opposite of elegant. Django models work with any database Django knows about.

» Having your data definitions stored in the database instead of in code means it's harder to keep your models under version control. Django does its best to support development best practices, which certainly includes the use of source control.

» Your application will need to work with the database design regardless of how you created it. If you created it using SQL, you'd have to repeat yourself. Don't Repeat Yourself.

» Writing Python is fun! Django does its best to keep you writing Python all the time.
First, though, we need to put this code somewhere where Django expects it. This is how you create an app; this'll create a package with a `models.py` and a `views.py` file. That code in the previous slide goes into `models.py`.

For simplicity, we’re going to keep this packages app inside the project (i.e. it’ll be `cheeserater.packages`), but there’s no reason that apps have to be part of a project. In fact, as you develop more Django sites, you’ll likely find that “projects” will usually just contain (via `INSTALLED_APPS`) pointers to apps that exist without any given project.

You’ve now created a model, but there’s one more step to getting this model registered with Django. In your `settings.py` file, you’ll see a list called `INSTALLED_APPS`. This list tells a given Django instance which apps should be considered “active” for that instance. This lets you have a large library of apps and select which ones you want enabled on a site-by-site basis.

Here’s that Package model definition again; this goes in the `models.py` file we just created.
The next step is to run `validate` to check that the model definition is OK. Looks like it is.

```
$ ./manage.py validate
@ errors found.
```

Run `syncdb` when you add new models and they'll be installed into the database.

```
$ ./manage.py syncdb
Creating table packages_package ...
```

Once you've created a model, Django automatically provides a high-level Python API for working with those models.

**Model API**
Although it's only a few lines of code, this actually does quite a bit. You can see how objects are created (Django objects are simply Python classes) and saved to the database (all Django objects have a `save()` method which saves the object to the database).

This is another point of philosophy in Django: it won’t hit the database unless you explicitly ask for it (hence requiring the `save()` call). This makes it always easy to read your code and figure out where the DB hits are.

Here we've got simple data retrieval. Every model gets an `objects` member which has functions for fetching content from the database.

You can get a taste of how the lookup API works with the `name__contains` business; that's a lookup for a link whose name attribute contains the string “Dj”.

Technically, this `objects` thing is a `Manager`. I won't really go into them too much in this tutorial, but now you'll recognize the term when you read the documentation.

Recall that earlier I claimed that metadata is one reason to define models in Python? So let’s do a bit of that.
__unicode__ is used any time Django wants to spit out a “string version” of the model, and it’s very useful.

Here we’ve added a bunch of metadata. None of it is required, but it’s all optional. This is another theme of Django: provide sensible defaults, but allow you to override them easily.

Of course, what’s the point of using a relational database if we don’t use relationships? Let’s add some.

**Relationships**

Here are — minus any metadata — the models for the classification models. Notice the ForeignKey, and notice it’s only defined in one place.

Also notice the related_name attribute there; we’ll talk about it shortly.
A number of things to notice here:

» Even though I didn’t explicitly tell Django about the relationship from Topic to Category, Django still fills it in for me. The name categories comes from the ForeignKey related_name attribute; if I hadn’t given that argument, Django would have created one automatically for me (called category_set).

» This object (categories) is a Manager just like Package.objects

Queries (filter() and friends) are lazy, so this doesn’t fetch all the objects and then just slice off the first but instead issues the correct limiting clause to the database. This means you can pass these QuerySets around and they don’t get evaluated until you need them.

» The ForeignKey we defined earlier can be used quite naturally as an attribute of the instance just like you’d expect. This does makes the correct DB lookup in the background, but it’s cached so it only gets performed once.

» Many-to-many relations work just as the reverse foreign key ones do. Also notice that I can assign M2M relations by assigning an iterable to the relation.

For my money, the automatic following of relationships is the best feature of Django’s DB API. When we created the model, we only needed to specify the ForeignKey and Django automatically constructed accessors on the “other side” of the relationship. We Didn’t have to Repeat Ourselves.
OK, so here’s an interesting challenge. Think for a few moments about the SQL you’d need to write for this, and then I’ll show the ORM code.

Again, my favorite part is the following of relationships in “both ways.”

Earlier I mentioned that these queries are “lazy.” This is what I’m talking about: this example performs exactly one SQL query, which happens when packages is evaluated (i.e. printed).
There’s a huge number of options and coolnesses that I can’t possibly cover in three hours, but these documents taken together cover every single possible thing the ORM layer is capable of:

- [http://www.djangobook.com/en/1.0/chapter05/](http://www.djangobook.com/en/1.0/chapter05/)
- [http://www.djangoproject.com/documentation/db_api/](http://www.djangoproject.com/documentation/db_api/)
- [http://www.djangoproject.com/documentation/models/](http://www.djangobook.com/en/1.0/chapter05/)

The first two are references for the model definition API and the database lookup API; the third link is to a large set of examples.

The final part of our exploration of Django models is the automatic admin interface.
There's one part of web development we've always hated: writing administration interfaces. Developing the parts of the site that the general public sees is always different and interesting, but the bits that the administrators use to modify the site always are the same. You've got to deal with authenticating users to make sure that they are allowed to edit the content, display and handle forms, deal with tricky validation issues... it's boring, and it's repetitive.

One of the oldest and most powerful parts of Django is the automatic admin interface. It hooks off of metadata in your model to provide a powerful and production-ready interface that content producers can immediately use to start adding content to the site.

Although Django can be — and these days is — used for anything, for us at the Journal-World the admin interface is the raison d'être of our love for Django. With it we can quickly define models and then turn them over to reporters and editors to fill out data.

We think the admin interface is the coolest part of Django -- and most Djangonauts agree -- but since not everyone actually needs it, it's an optional piece. That means there are three steps you'll need to follow to activate the admin interface.
First, we need to “register” models so that the admin will edit them -- not all models can (or should) be editable by admin users. You do that with this `admin.site.register()` call; it takes the model to register as its required argument.

The code can live anywhere you like. However, it needs to somehow get run so that the register call gets executed. By convention, then, we put these calls in your app’s `admin.py` file and we’ll “auto-discover” these admin files so that the models get registered.

There are a lot of options you can give your admin when you register a model; here are just a few. This is the basic idiom of how you register models with the admin and include options.

Next, we need to install the admin models (remember to run `syncdb`!)

This is also a good time to run `manage.py createsuperuser` if you didn’t create a superuser the first time around.
from django.conf.urls.defaults import *  
from django.contrib import admin  

admin.autodiscover()  

urlpatterns = patterns('',  
    ('^admin/(.*)', admin.site.root,  
)

Finally, uncomment the parts in urls.py that it tells you to uncomment. This autodiscover() call picks up the admin.py file we created earlier.

... and run the development server to play with the admin.

$ ./manage.py runserver
Live demo! For those who don’t have the dubious pleasure of watching my demo, my rambling will probably involve some or all of these nifty features:

» The admin has a fairly powerful user/group/permission system. We don’t see it here since we’re working as a superuser, but other users can easily have their access restricted.

» Each object given an Admin declaration shows up on the main index page. Links to add and change objects lead to two pages we refer to as object "change lists" and "edit forms".

» Change lists are essentially index pages of objects in the system. Admin options that control which fields appear on these lists and the appearance of extra features like date drill downs, search fields, and filter interfaces.

» Edit forms are used to edit existing objects and create new ones. Each field defined in your model appears here, and you’ll notice that fields of different types get different widgets. The admin also handles input validation for you; try leaving a required field blank, or putting an invalid value into a field.

» When editing an existing object, you’ll notice a “history” link in the upper-right. Every change made through the admin is logged, and you can examine this log by clicking the history.

» Deletions in the admin cascade. For example, if you go to delete a Classifier, you’ll see that the related PackageClassification objects are scheduled for deletion as well.
More...


Views

This is where it gets really interesting.

Up until this point, we've been dealing with how Django simplifies the boring, annoying, repetitive tasks every web developer faces. Now that we've gotten them out of the way, we can finally deal with the exciting, fun part of web development: creating the public facing views.

I refer to views you write yourself as "public views" to differentiate them from the built-in admin interface, but really these "public" views could be anywhere on the spectrum of private-only-for-a-single-user to completely public world-accessibly websites.

This is also the point where Django philosophy of "getting out of your way" comes into play: when writing views, you're pretty much left to your own devices as Django tries to be as unobtrusive as possible.

What’s a view?

A view is a "type" of Web page in your Django application that generally serves a specific function and has a specific template. For example, on our example site we’ll eventually have views for

- The homepage
- An index of packages
- An individual package’s detail page
- A user page, showing which packages they’ve voted on
- And possibly more...
In essence, a view is responsible for pulling together all the information accessible at a given URL. It's where you look up data from a database, read it in from a file, aggregate or summarize it, etc. Note that a view is not where you define how this data looks; that's up to a template, which we'll cover a bit later.

Wait, I thought we were working on views?

Like it or not, URLs are a part of your application design. Django forces you to think about them, because these are ugly.

Why do I (the reader) need to know that you’re using PHP? Or the ID of your story? In the bottom case, I could even be signaling a security problem just with my extension!

And this is just stupid.
Ahhh... that’s much much nicer. That’s what we’ll use.

When a user requests a Django-powered page, Django looks at the `ROOT_URLCONF` setting, which contains a Python path to a module. Django loads that module and looks for some urlpatterns.

```
ROOT_URLCONF = "cheeserater.urls"
```

The first step of writing views is to design your URL structure. You do this by creating a Python module, called a `URLconf`. URLconf's are how Django associates a given URL with given Python code.

A urlpattern is simply a list of regular expressions and function names. Django walks down the list until it finds a regex that matches; it then calls the provided function.

There’s a small problem here. Why’d we go to the trouble of putting the packages into a separate app if we’re just going to couple it the the project URL patterns? What if on a future project we want to reuse the packages app but stick it under, say, `/code`?

So, we’ll make a quick change to decouple the URLs from the project.
This is worth a review. When somebody requests a page from your Web site -- say, /packages/Unipath/, this is what happens:

» Django will load cheeserater.urls Python module, because it's pointed to by the ROOT_URLCONF setting.

» In cheeserater.urls, the patterns are evaluated in order. The last pattern matches.

» Django then calls the function you’ve given as the callback. Here it’s package_detail.

» The view gets a "request" object as the first parameter. Here, the package_name param comes from the (\?P<package_name>\.*\) part in the regex. Using parenthesis around a pattern "captures" the text and sends it as an argument to the view function; the \?P<package_name> defines the name that will be used to identify the matched pattern.

» The view returns a response, which is passed back to the browser (more on this next...)

The URL dispatch engine is pretty simple (I think), but there’s comprehensive documentation at http://www.djangoproject.com/documentation/url_dispatch/
So here’s perhaps the simplest possible view we could write for the package_list view. It shows one important thing, though: every single view takes a request (and possibly other options) and returns an `HttpResponse` object.

OK, a little more complex here; now we’re getting some data from the database and rendering it.

However, this should be pretty obviously bad: we’re building the page’s design right into the view, and setting ourselves up for disaster. If you want to change the HTML, you’d need to edit the Python, and you shouldn’t have to do that.

A little better. Now we’re loading a template and passing in some stuff to it (we’ll gloss over the template for now and come back to it later on).

However, in the interest of keeping code short, let’s modify this to use a shortcut instead.
This does exactly the same thing as the last view; it’s just nice and concise.

I’ll quickly point out a problem here: there’s about 4000 packages in the cheesehop; that’s one crazy-long list! We really need pagination here, and we’ll come back to that later when we use a generic view to render this page.

So we’ve got a view now, but no matching template. Let’s work on that.

```python
from django.shortcuts import render_to_response
from cheeaserer.packages.models import Package

def package_index(request):
    ps = Package.objects.order_by("name")
    return render_to_response(
        "packages/index.html",
        {"package_list" : ps}
    )
```
What’s a template?

If a view describes what data is displayed on a given page, the template describes how that data looks. This distinction is important since it will easily let you change how your site looks without having to change how it behaves. Most web development teams already have different people who develop the code who design the templates; this division in Django strives to match that natural division of labor.

Django’s template language is designed to strike a balance between power and ease. It’s designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as Smarty (http://smarty.php.net) or CheetahTemplate (http://www.cheetahtemplate.org/), you should feel right at home with Django’s templates.

We have a deep love for Django’s template language, but not everyone shares our passion. Luckily, like most of Django, there’s nothing that binds you to using Django’s template engine instead of the one of your choice. You can simply write views that render content using your favorite engine instead.

Django’s template language has particularly been designed to be designer-friendly. We think most programmers -- us included -- have the design sense of a color-blind three-year-old, so we prefer to leave the designing to the professionals by giving them a template language they can understand.

Here’s a nice, complete, valid template we could use for the “recent links” view.

I’ll go over the salient features.

First thing to notice is that templates are simply text files. Much of the time you’ll be using them to generate HTML, but you could generate any text-based format (XML, CSV, YAML, whatever).
These are variables. When the template engine encounters a variable, it evaluates it and replaces it with the value of the variable.

The variables are evaluated against the context that we created in the view earlier.

Because template authors shouldn’t need to know Python, the dot in a template tries dictionary lookups, attribute looks, list-index lookups, and will call callables that take no arguments.

Finally, if none of those work, the contents of the setting `TEMPLATE_STRING_IF_INVALID` will be returned (which defaults to the empty string).

This is a filter.

Think of filters as a pipe: a variable goes in one end, and something else comes out the other.

Going with the pipe analogy, filters can be “chained” — the output of one passed onto the next. The above is a common idiom for dealing with user-submitted text; it escapes potentially dangerous HTML, and then turns linebreaks into `<p>` tags.
Some filters take arguments, which are given after a colon in double quotes. This example truncates the text to 30 words.

```html
{{ text|truncatetwords:"30" }}
```

These are tags. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags; `{% for %}` is one of those tags. Ending tags are always of the form `{% end<tag> %}`.

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

The most powerful -- and thus the most complex -- part of Django's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines blocks that child templates can override.

Understanding inheritance begins with understanding the problem it solves. The template we designed previous looks just fine, but what happens when we go to create a template for the year archive? We'd start by duplicating all the DOCTYPE, <head>, etc. crap -- and that's a major pain. Now imagine you’ve got thousands of templates, and it’s clear you need a way of keeping all the repeated code in one place.
This template, which we'll call `base.html`, defines a simple HTML skeleton document that we'll use for all the pages on the site. It's the job of "child" templates to fill the empty blocks with content.

In this example, the `{% block %}` tag defines three blocks that child templates can fill in. All the block tag does is to tell the template engine that a child template may override those portions of the template.

So now we can go back to the package index template and "extend" the base template we just created.

The `{% extends %}` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent -- in this case, `base.html`.

At that point, the template engine will notice the `{% block %}` tags in `base.html` and replace those blocks with the contents of the child template. So the title we've defined here will be used, as will all the template code to loop over and display the package list. The skeleton from the base template will be filled in with the content from the child one.

Note that since the child template didn't define the footer block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.
One common way of using inheritance is a three-level approach:

» Create a `base.html` template that holds the main look-and-feel of your site.

» Create a `base_SECTION.html` template for each "section" of your site. For example, `base_news.html`, `base_sports.html`. These templates all extend `base.html` and include section-specific styles/design.

» Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Why use template inheritance? Mostly because it makes it incredibly easy to design sites. You can swap out base templates for an entirely new look with ease.
Here are some tips for working with inheritance:

» If you use {% extends %} in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.

» More {% block %} tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.

» If you find yourself duplicating content in a number of templates, it probably means you should move that content to a {% block %} in a parent template.

» If you need to get the content of the block from the parent template, the {{ block.super }} variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it.

For more on templates from the point of view of a template author, see http://www.djangoproject.com/documentation/templates/. There’s also a guide to the templates for Python developers at http://www.djangoproject.com/documentation/templates_python/. There’s also the Django Book: http://djangobook.com/en/1.0/chapter04/.
There’s so much more to Django...

That’s all I got, but that’s far from all there is.

I hope you’ll keep hacking on Django. If you’re at all interested, I recommend joining the users group; I really take pride in the quality of our community.

And feel free to email me directly any time.