Chapter 8: Advanced Views and URLconfs

In Chapter 3, we explained the basics of Django view functions and URLconfs. This chapter goes into more detail about advanced functionality in those two pieces of the framework.

URLconf Tricks

There's nothing "special" about URLconfs – like anything else in Django, they're just Python code. You can take advantage of this in several ways, as described in the sections that follow.

Streamlining Function Imports

Consider this URLconf, which builds on the example in Chapter 3:

```
from django.conf.urls.defaults import *

from mysite.views import hello, current_datetime,
hours_ahead


urlpatterns = patterns('',

    (r'^hello/$', hello),

    (r'^time/$', current_datetime),

    (r'^time/plus/(\d{1,2})/$', hours_ahead),

)
```

As explained in Chapter 3, each entry in the URLconf includes its associated view function, passed directly as a function object. This means it's necessary to import the view functions at the top of the module.

But as a Django application grows in complexity, its URLconf grows, too, and keeping those imports can be tedious to manage. (For each new view function, you have to remember to import it, and the import statement tends to get overly long if you use this approach.) It's possible to avoid that tedium by importing the views module itself. This example URLconf is equivalent to the previous one:

```
from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',
```

```
    (r'^hello/$', views.hello),
    (r'^time/$', views.current_datetime),
    (r'^time/plus/(d{1,2})/$', views.hours_ahead),
)
```

Django offers another way of specifying the view function for a particular pattern in the URLconf: you can pass a string containing the module name and function name rather than the function object itself. Continuing the ongoing example:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(d{1,2})/$',
'mysite.views.hours_ahead'),
)
```

(Note the quotes around the view names. We're using 'mysite.views.current_datetime' – with quotes – instead of mysite.views.current_datetime.)

Using this technique, it's no longer necessary to import the view functions; Django automatically imports the appropriate view function the first time it's needed, according to the string describing the name and path of the view function.

A further shortcut you can take when using the string technique is to factor out a common "view prefix." In our URLconf example, each of the view strings starts with 'mysite.views', which is redundant to type. We can factor out that common prefix and pass it as the first argument to patterns(), like this:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
```

```
    (r'^time/plus/(d{1,2})/$', 'hours_ahead'),

)
```

Note that you don't put a trailing dot (".") in the prefix, nor do you put a leading dot in the view strings. Django puts those in automatically.

With these two approaches in mind, which is better? It really depends on your personal coding style and needs.

Advantages of the string approach are as follows:

- It's more compact, because it doesn't require you to import the view functions.
- It results in more readable and manageable URLconfs if your view functions are spread across several different Python modules.

Advantages of the function object approach are as follows:

- It allows for easy "wrapping" of view functions. See the section "Wrapping View Functions" later in this chapter.
- It's more "Pythonic" – that is, it's more in line with Python traditions, such as passing functions as objects.

Both approaches are valid, and you can even mix them within the same URLconf. The choice is yours.

Using Multiple View Prefixes

In practice, if you use the string technique, you'll probably end up mixing views to the point where the views in your URLconf won't have a common prefix. However, you can still take advantage of the view prefix shortcut to remove duplication. Just add multiple patterns() objects together, like this:

Old:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('',

    (r'^hello/$', 'mysite.views.hello'),

    (r'^time/$', 'mysite.views.current_datetime'),

    (r'^time/plus/(\d{1,2})/$',
'mysite.views.hours_ahead'),

    (r'^tag/(\w+)/$', 'weblog.views.tag'),
```

```
)
```

New:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
    (r'^time/plus/(\d{1,2})/$', 'hours_ahead'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

All the framework cares about is that there's a module-level variable called urlpatterns. This variable can be constructed dynamically, as we do in this example. We should specifically point out that the objects returned by patterns() can be added together, which is something you might not have expected.

Special-Casing URLs in Debug Mode

Speaking of constructing urlpatterns dynamically, you might want to take advantage of this technique to alter your URLconf's behavior while in Django's debug mode. To do this, just check the value of the DEBUGsetting at runtime, like so:

```
from django.conf import settings
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^$', views.homepage),
```

```
    (r'^(\d{4})/([a-z]{3})/$', views.archive_month),
)


if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^debuginfo/$', views.debug),
    )
```

In this example, the URL /debuginfo/ will only be available if your DEBUG setting is set to True.

Using Named Groups

In all of our URLconf examples so far, we've used simple, *non-named* regular expression groups – that is, we put parentheses around parts of the URL we wanted to capture, and Django passes that captured text to the view function as a positional argument. In more advanced usage, it's possible to use *named* regular expression groups to capture URL bits and pass them as *keyword* arguments to a view.

Keyword Arguments vs. Positional Arguments

A Python function can be called using keyword arguments or positional arguments – and, in some cases, both at the same time. In a keyword argument call, you specify the names of the arguments along with the values you're passing. In a positional argument call, you simply pass the arguments without explicitly specifying which argument matches which value; the association is implicit in the arguments' order.

For example, consider this simple function:

```
def sell(item, price, quantity):
    print "Selling %s unit(s) of %s at %s" % (quantity, item, price)
```

To call it with positional arguments, you specify the arguments in the order in which they're listed in the function definition:

```
sell('Socks', '$2.50', 6)
```

To call it with keyword arguments, you specify the names of the arguments along with the values. The following statements are equivalent:

```
sell(item='Socks', price='$2.50', quantity=6)
```

```
sell(item='Socks', quantity=6, price='$2.50')

sell(price='$2.50', item='Socks', quantity=6)

sell(price='$2.50', quantity=6, item='Socks')

sell(quantity=6, item='Socks', price='$2.50')

sell(quantity=6, price='$2.50', item='Socks')
```

Finally, you can mix keyword and positional arguments, as long as all positional arguments are listed before keyword arguments. The following statements are equivalent to the previous examples:

```
sell('Socks', '$2.50', quantity=6)

sell('Socks', price='$2.50', quantity=6)

sell('Socks', quantity=6, price='$2.50')
```

In Python regular expressions, the syntax for named regular expression groups is (?P<name>pattern), where name is the name of the group and pattern is some pattern to match.

Here's an example URLconf that uses non-named groups:

```
from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^articles/(\d{4})/$', views.year_archive),

    (r'^articles/(\d{4})/(\d{2})/$',
views.month_archive),

)
```

Here's the same URLconf, rewritten to use named groups:

```
from django.conf.urls.defaults import *

from mysite import views
```

```
urlpatterns = patterns('',

    (r'^articles/(?P<year>\d{4})/$',
views.year_archive),

    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$',
views.month_archive),

)
```

This accomplishes exactly the same thing as the previous example, with one subtle difference: the captured values are passed to view functions as keyword arguments rather than positional arguments.

For example, with non-named groups, a request to /articles/2006/03/ would result in a function call equivalent to this:

```
month_archive(request, '2006', '03')
```

With named groups, though, the same request would result in this function call:

```
month_archive(request, year='2006', month='03')
```

In practice, using named groups makes your URLconfs slightly more explicit and less prone to argument-order bugs – and you can reorder the arguments in your views' function definitions. Following the preceding example, if we wanted to change the URLs to include the month *before* the year, and we were using non-named groups, we'd have to remember to change the order of arguments in the month_archive view. If we were using named groups, changing the order of the captured parameters in the URL would have no effect on the view.

Of course, the benefits of named groups come at the cost of brevity; some developers find the named-group syntax ugly and too verbose. Still, another advantage of named groups is readability, especially by those who aren't intimately familiar with regular expressions or your particular Django application. It's easier to see what's happening, at a glance, in a URLconf that uses named groups.

Understanding the Matching/Grouping Algorithm

A caveat with using named groups in a URLconf is that a single URLconf pattern cannot contain both named and non-named groups. If you do this, Django won't throw any errors, but you'll probably find that your URLs aren't matching as you expect. Specifically, here's the algorithm the URLconf parser follows, with respect to named groups vs. non-named groups in a regular expression:

- If there are any named arguments, it will use those, ignoring non-named arguments.
- Otherwise, it will pass all non-named arguments as positional arguments.
- In both cases, it will pass any extra options as keyword arguments. See the next section for more information.

Passing Extra Options to View Functions

Sometimes you'll find yourself writing view functions that are quite similar, with only a few small differences. For example, say you have two views whose contents are identical except for the template they use:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^foo/$', views.foo_view),

    (r'^bar/$', views.bar_view),

)


# views.py


from django.shortcuts import render

from mysite.models import MyModel


def foo_view(request):

    m_list = MyModel.objects.filter(is_new=True)

    return render(request, 'template1.html', {'m_list':
m_list})


def bar_view(request):

    m_list = MyModel.objects.filter(is_new=True)
```

```
    return render(request, 'template2.html', {'m_list':
m_list})
```

We're repeating ourselves in this code, and that's inelegant. At first, you may think to remove the redundancy by using the same view for both URLs, putting parentheses around the URL to capture it, and checking the URL within the view to determine the template, like so:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^(foo)/$', views.foobar_view),

    (r'^(bar)/$', views.foobar_view),

)


# views.py


from django.shortcuts import render

from mysite.models import MyModel


def foobar_view(request, url):

    m_list = MyModel.objects.filter(is_new=True)

    if url == 'foo':

        template_name = 'template1.html'

    elif url == 'bar':

        template_name = 'template2.html'
```

```
    return render(request, template_name, {'m_list':
m_list})
```

The problem with that solution, though, is that it couples your URLs to your code. If you decide to rename/foo/ to /fooey/, you'll have to remember to change the view code.

The elegant solution involves an optional URLconf parameter. Each pattern in a URLconf may include a third item: a dictionary of keyword arguments to pass to the view function.

With this in mind, we can rewrite our ongoing example like this:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^foo/$', views.foobar_view, {'template_name':
'template1.html'}),

    (r'^bar/$', views.foobar_view, {'template_name':
'template2.html'}),

)


# views.py


from django.shortcuts import render

from mysite.models import MyModel


def foobar_view(request, template_name):

    m_list = MyModel.objects.filter(is_new=True)

    return render(request, template_name, {'m_list':
m_list})
```

As you can see, the URLconf in this example specifies template_name in the URLconf. The view function treats it as just another parameter.

This extra URLconf options technique is a nice way of sending additional information to your view functions with minimal fuss. As such, it's used by a couple of Django's bundled applications, most notably its generic views system, which we cover in Chapter 11.

The following sections contain a couple of ideas on how you can use the extra URLconf options technique in your own projects.

Faking Captured URLconf Values

Say you have a set of views that match a pattern, along with another URL that doesn't fit the pattern but whose view logic is the same. In this case, you can "fake" the capturing of URL values by using extra URLconf options to handle that extra URL with the same view.

For example, you might have an application that displays some data for a particular day, with URLs such as these:

```
/mydata/jan/01/

/mydata/jan/02/

/mydata/jan/03/

# ...

/mydata/dec/30/

/mydata/dec/31/
```

This is simple enough to deal with – you can capture those in a URLconf like this (using named group syntax):

```
urlpatterns = patterns('',

    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$',
views.my_view),

)
```

And the view function signature would look like this:

```
def my_view(request, month, day):

    # ....
```

This approach is straightforward – it's nothing you haven't seen before. The trick comes in when you want to add another URL that uses my_view but whose URL doesn't include a month and/or day.

For example, you might want to add another URL, /mydata/birthday/, which would be equivalent to/mydata/jan/06/. You can take advantage of extra URLconf options like so:

```
urlpatterns = patterns('',

    (r'^mydata/birthday/$', views.my_view, {'month':
'jan', 'day': '06'}),

    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$',
views.my_view),

)
```

The cool thing here is that you don't have to change your view function at all. The view function only cares that it *gets* month and day parameters – it doesn't matter whether they come from the URL capturing itself or extra parameters.

Making a View Generic

It's good programming practice to "factor out" commonalities in code. For example, with these two Python functions:

```
def say_hello(person_name):

    print 'Hello, %s' % person_name


def say_goodbye(person_name):

    print 'Goodbye, %s' % person_name
```

we can factor out the greeting to make it a parameter:

```
def greet(person_name, greeting):

    print '%s, %s' % (greeting, person_name)
```

You can apply this same philosophy to your Django views by using extra URLconf parameters.

With this in mind, you can start making higher-level abstractions of your views. Instead of thinking to yourself, "This view displays a list of Event objects," and "That view displays a list of BlogEntry objects," realize they're both specific cases of "A view that displays a list of objects, where the type of object is variable."

Take this code, for example:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^events/$', views.event_list),

    (r'^blog/entries/$', views.entry_list),

)


# views.py


from django.shortcuts import render

from mysite.models import Event, BlogEntry


def event_list(request):

    obj_list = Event.objects.all()

    return render(request, 'mysite/event_list.html',
{'event_list': obj_list})


def entry_list(request):

    obj_list = BlogEntry.objects.all()

    return render(request,
'mysite/blogentry_list.html', {'entry_list': obj_list})
```

The two views do essentially the same thing: they display a list of objects. So let’s factor out the type of object they’re displaying:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import models, views


urlpatterns = patterns('',

    (r'^events/$', views.object_list, {'model':
models.Event}),

    (r'^blog/entries/$', views.object_list, {'model':
models.BlogEntry}),

)


# views.py


from django.shortcuts import render


def object_list(request, model):

    obj_list = model.objects.all()

    template_name = 'mysite/%s_list.html' %
model.__name__.lower()

    return render(request, template_name,
{'object_list': obj_list})
```

With those small changes, we suddenly have a reusable, model-agnostic view! From now on, anytime we need a view that lists a set of objects, we can simply reuse this object_list view rather than writing view code. Here are a couple of notes about what we did:

- We're passing the model classes directly, as the model parameter. The dictionary of extra URLconf options can pass any type of Python object – not just strings.
- The model.objects.all() line is an example of *duck typing*: "If it walks like a duck and talks like a duck, we can treat it like a duck." Note the code doesn't know what type of object model is; the only requirement is that model have an objects attribute, which in turn has an all() method.

- We're using model.__name__.lower() in determining the template name. Every Python class has a__name__ attribute that returns the class name. This feature is useful at times like this, when we don't know the type of class until runtime. For example, the BlogEntry class's __name__ is the string'BlogEntry'.
- In a slight difference between this example and the previous example, we're passing the generic variable name object_list to the template. We could easily change this variable name to beblogentry_list or event_list, but we've left that as an exercise for the reader.

  Because database-driven Web sites have several common patterns, Django comes with a set of "generic views" that use this exact technique to save you time. We cover Django's built-in generic views in Chapter 11.

Giving a View Configuration Options

If you're distributing a Django application, chances are that your users will want some degree of configuration. In this case, it's a good idea to add hooks to your views for any configuration options you think people may want to change. You can use extra URLconf parameters for this purpose.

A common bit of an application to make configurable is the template name:

```
def my_view(request, template_name):
    var = do_something()
    return render(request, template_name, {'var': var})
```

Understanding Precedence of Captured Values vs. Extra Options

When there's a conflict, extra URLconf parameters get precedence over captured parameters. In other words, if your URLconf captures a named-group variable and an extra URLconf parameter includes a variable with the same name, the extra URLconf parameter value will be used.

For example, consider this URLconf:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id':
3}),
)
```

Here, both the regular expression and the extra dictionary include an id. The hard-coded id gets precedence. That means any request (e.g., /mydata/2/ or /mydata/432432/) will be treated as if id is set to3, regardless of the value captured in the URL.

Astute readers will note that in this case, it's a waste of time and typing to capture the id in the regular expression, because its value will always be overridden by the dictionary's value. That's correct; we bring this up only to help you avoid making the mistake.

Using Default View Arguments

Another convenient trick is to specify default parameters for a view's arguments. This tells the view which value to use for a parameter by default if none is specified.

An example:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^blog/$', views.page),

    (r'^blog/page(?P<num>\d+)/$', views.page),

)


# views.py


def page(request, num='1'):

    # Output the appropriate page of blog entries,
according to num.

    # ...
```

Here, both URL patterns point to the same view – views.page – but the first pattern doesn't capture anything from the URL. If the first pattern matches, the page() function will use its default argument for num,'1'. If the second pattern matches, page() will use whatever num value was captured by the regular expression.

(Note that we've been careful to set the default argument's value to the *string* '1', not the integer 1. That's for consistency, because any captured value for num will always be a string.)

It's common to use this technique in conjunction with configuration options, as explained earlier. This example makes a slight improvement to the example in the "Giving a View Configuration Options" section by providing a default value for template_name:

```
def my_view(request,
template_name='mysite/my_view.html'):

    var = do_something()

    return render(request, template_name, {'var': var})
```

Special-Casing Views

Sometimes you'll have a pattern in your URLconf that handles a large set of URLs, but you'll need to special-case one of them. In this case, take advantage of the linear way a URLconf is processed and put the special case first.

For example, you can think of the "add an object" pages in Django's admin site as represented by a URLpattern like this:

```
urlpatterns = patterns('',

    # ...

    ('^([^/]+)/([^/]+)/add/$', views.add_stage),

    # ...

)
```

This matches URLs such as /myblog/entries/add/ and /auth/groups/add/. However, the "add" page for a user object (/auth/user/add/) is a special case – it doesn't display all of the form fields, it displays two password fields, and so forth. We *could* solve this problem by special-casing in the view, like so:

```
def add_stage(request, app_label, model_name):

    if app_label == 'auth' and model_name == 'user':

        # do special-case code

    else:

        # do normal code
```

but that's inelegant for a reason we've touched on multiple times in this chapter: it puts URL logic in the view. As a more elegant solution, we can take advantage of the fact that URLconfs are processed in order from top to bottom:

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', views.user_add_stage),
    ('^([^/]+)/([^/]+)/add/$', views.add_stage),
    # ...
)
```

With this in place, a request to /auth/user/add/ will be handled by the user_add_stage view. Although that URL matches the second pattern, it matches the top one first. (This is short-circuit logic.)

Capturing Text in URLs

Each captured argument is sent to the view as a plain Python Unicode string, regardless of what sort of match the regular expression makes. For example, in this URLconf line:

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

the year argument to views.year_archive() will be a string, not an integer, even though \d{4} will only match integer strings.

This is important to keep in mind when you're writing view code. Many built-in Python functions are fussy (and rightfully so) about accepting only objects of a certain type. A common error is to attempt to create adatetime.date object with string values instead of integer values:

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
    ...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

Translated to a URLconf and view, the error looks like this:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    (r'^articles/(\d{4})/(\d{2})/(\d{2})/$',
views.day_archive),

)


# views.py


import datetime


def day_archive(request, year, month, day):

    # The following statement raises a TypeError!

    date = datetime.date(year, month, day)
```

Instead, day_archive() can be written correctly like this:

```
def day_archive(request, year, month, day):

    date = datetime.date(int(year), int(month),
int(day))
```

Note that int() itself raises a ValueError when you pass it a string that is not composed solely of digits, but we're avoiding that error in this case because the regular expression in our URLconf has ensured that only strings containing digits are passed to the view function.

Determining What the URLconf Searches Against

When a request comes in, Django tries to match the URLconf patterns against the requested URL, as a Python string. This does not include GET or POST parameters, or the domain name. It also does not include the leading slash, because every URL has a leading slash.

For example, in a request to http://www.example.com/myapp/, Django will try to match myapp/. In a request to http://www.example.com/myapp/?page=3, Django will try to match myapp/.

The request method (e.g., POST, GET) is *not* taken into account when traversing the URLconf. In other words, all request methods will be routed to the same function for the same URL. It's the responsibility of a view function to perform branching based on request method.

Higher-Level Abstractions of View Functions

And speaking of branching based on request method, let's take a look at how we might build a nice way of doing that. Consider this URLconf/view layout:

```
# urls.py


from django.conf.urls.defaults import *

from mysite import views


urlpatterns = patterns('',

    # ...

    (r'^somepage/$', views.some_page),

    # ...

)


# views.py


from django.http import Http404, HttpResponseRedirect

from django.shortcuts import render


def some_page(request):

    if request.method == 'POST':

        do_something_for_post()
```

```
        return HttpResponseRedirect('/someurl/')
    elif request.method == 'GET':
        do_something_for_get()
        return render(request, 'page.html')
    else:
        raise Http404()
```

In this example, the some_page() view's handling of POST vs. GET requests is quite different. The only thing they have in common is a shared URL: /somepage/. As such, it's kind of inelegant to deal with both POST andGET in the same view function. It would be nice if we could have two separate view functions – one handlingGET requests and the other handling POST – and ensuring each one was only called when appropriate.

We can do that by writing a view function that delegates to other views, either before or after executing some custom logic. Here's an example of how this technique could help simplify our some_page() view:

```
# views.py


from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render


def method_splitter(request, GET=None, POST=None):
    if request.method == 'GET' and GET is not None:
        return GET(request)
    elif request.method == 'POST' and POST is not None:
        return POST(request)
    raise Http404


def some_page_get(request):
    assert request.method == 'GET'
```

```
    do_something_for_get()
    return render(request, 'page.html')


def some_page_post(request):
    assert request.method == 'POST'
    do_something_for_post()
    return HttpResponseRedirect('/someurl/')


# urls.py


from django.conf.urls.defaults import *
from mysite import views


urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.method_splitter, {'GET':
views.some_page_get, 'POST': views.some_page_post}),
    # ...
)
```

Let's go through what this does:

- We've written a new view, method_splitter(), that delegates to other views based onrequest.method. It looks for two keyword arguments, GET and POST, which should be *view functions*. Ifrequest.method is 'GET', then it calls the GET view. If request.method is 'POST', then it calls the POSTview. If request.method is something else (HEAD, etc.), or if GET or POST were not supplied to the function, then it raises an Http404.
- In the URLconf, we point /somepage/ at method_splitter() and pass it extra arguments – the view functions to use for GET and POST, respectively.

- Finally, we've split the some_page() view into two view functions – some_page_get() andsome_page_post(). This is much nicer than shoving all of that logic into a single view.

  Note that these view functions technically no longer have to check request.method, becausemethod_splitter() does that. (By the time some_page_post() is called, for example, we can be confident request.method is 'post'.) Still, just to be safe, and also to serve as documentation, we stuck in an assert that makes sure request.method is what we expect it to be.

  Now we have ourselves a nice, generic view function that encapsulates the logic of delegating a view byrequest.method. Nothing about method_splitter() is tied to our specific application, of course, so we can reuse it in other projects.

  But, while we're at it, there's one way to improve on method_splitter(). As it's written, it assumes that theGET and POST views take no arguments other than request. What if we wanted to use method_splitter()with views that, for example, capture text from URLs, or take optional keyword arguments themselves?

  To do that, we can use a nice Python feature: variable arguments with asterisks. We'll show the example first, then explain it:

```
def method_splitter(request, *args, **kwargs):
    get_view = kwargs.pop('GET', None)
    post_view = kwargs.pop('POST', None)
    if request.method == 'GET' and get_view is not
None:
        return get_view(request, *args, **kwargs)
    elif request.method == 'POST' and post_view is not
None:
        return post_view(request, *args, **kwargs)
    raise Http404
```

  Here, we've refactored method_splitter() to remove the GET and POST keyword arguments, in favor of*args and **kwargs (note the asterisks). This is a Python feature that allows a function to accept a dynamic, arbitrary number of arguments whose names aren't known until runtime. If you put a single asterisk in front of a parameter in a function definition, any *positional* arguments to that function will be rolled up into a single tuple. If you put two asterisks in front of a parameter in a function definition, any *keyword* arguments to that function will be rolled up into a single dictionary.

  For example, with this function:

```
def foo(*args, **kwargs):
```

```
    print "Positional arguments are:"
    print args
    print "Keyword arguments are:"
    print kwargs
```

Here's how it would work:

```
>>> foo(1, 2, 3)
Positional arguments are:
(1, 2, 3)
Keyword arguments are:
{}
>>> foo(1, 2, name='Adrian', framework='Django')
Positional arguments are:
(1, 2)
Keyword arguments are:
{'framework': 'Django', 'name': 'Adrian'}
```

Bringing this back to method_splitter(), you can see we're using *args and **kwargs to accept *any* arguments to the function and pass them along to the appropriate view. But before we do that, we make two calls to kwargs.pop() to get the GET and POST arguments, if they're available. (We're using pop() with a default value of None to avoid KeyError if one or the other isn't defined.)

Wrapping View Functions

Our final view trick takes advantage of an advanced Python technique. Say you find yourself repeating a bunch of code throughout various views, as in this example:

```
def my_view1(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
```

```
    return render(request, 'template1.html')


def my_view2(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render(request, 'template2.html')


def my_view3(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render(request, 'template3.html')
```

Here, each view starts by checking that request.user is authenticated – that is, the current user has successfully logged into the site – and redirects to /accounts/login/ if not. (Note that we haven't yet covered request.user – Chapter 14 does – but, as you might imagine, request.user represents the current user, either logged-in or anonymous.)

It would be nice if we could remove that bit of repetitive code from each of these views and just mark them as requiring authentication. We can do that by making a view wrapper. Take a moment to study this:

```
def requires_login(view):
    def new_view(request, *args, **kwargs):
        if not request.user.is_authenticated():
            return
HttpResponseRedirect('/accounts/login/')
        return view(request, *args, **kwargs)
    return new_view
```

This function, requires_login, takes a view function (view) and returns a new view function (new_view). The new function, new_view is defined *within* requires_login and handles the logic of checkingrequest.user.is_authenticated() and delegating to the original view (view).

Now, we can remove the if not request.user.is_authenticated() checks from our views and simply wrap them with requires_login in our URLconf:

```
from django.conf.urls.defaults import *

from mysite.views import requires_login, my_view1,
my_view2, my_view3


urlpatterns = patterns('',

    (r'^view1/$', requires_login(my_view1)),

    (r'^view2/$', requires_login(my_view2)),

    (r'^view3/$', requires_login(my_view3)),

)
```

This has the same effect as before, but with less code redundancy. Now we've created a nice, generic function – requires_login() that we can wrap around any view in order to make it require login.

Including Other URLconfs

If you intend your code to be used on multiple Django-based sites, you should consider arranging your URLconfs in such a way that allows for "including."

At any point, your URLconf can "include" other URLconf modules. This essentially "roots" a set of URLs below other ones. For example, this URLconf includes other URLconfs:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('',

    (r'^weblog/', include('mysite.blog.urls')),

    (r'^photos/', include('mysite.photos.urls')),

    (r'^about/$', 'mysite.views.about'),

)
```

(We saw this before in Chapter 6, when we introduced the Django admin site. The admin site has its own URLconf that you merely include() within yours.)

There's an important gotcha here: the regular expressions in this example that point to an include() do*not* have a $ (end-of-string match character) but *do* include a trailing slash. Whenever Django encountersinclude(), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Continuing this example, here's the URLconf mysite.blog.urls:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('',

    (r'^(\d\d\d\d)/$',
'mysite.blog.views.year_detail'),

    (r'^(\d\d\d\d)/(\d\d)/$',
'mysite.blog.views.month_detail'),

)
```

With these two URLconfs, here's how a few sample requests would be handled:

- /weblog/2007/: In the first URLconf, the pattern r'^weblog/' matches. Because it is an include(), Django strips all the matching text, which is 'weblog/' in this case. The remaining part of the URL is2007/, which matches the first line in the mysite.blog.urls URLconf.
- /weblog//2007/ (with two slashes): In the first URLconf, the pattern r'^weblog/' matches. Because it is an include(), Django strips all the matching text, which is 'weblog/' in this case. The remaining part of the URL is /2007/ (with a leading slash), which does not match any of the lines in themysite.blog.urls URLconf.
- /about/: This matches the view mysite.views.about in the first URLconf, demonstrating that you can mix include() patterns with non-include() patterns.

How Captured Parameters Work with include()

An included URLconf receives any captured parameters from parent URLconfs, for example:

```
# root urls.py


from django.conf.urls.defaults import *


urlpatterns = patterns('',
```

```
    (r'^(?P<username>\w+)/blog/',
include('foo.urls.blog')),

)


# foo/urls/blog.py


from django.conf.urls.defaults import *


urlpatterns = patterns('',

    (r'^$', 'foo.views.blog_index'),

    (r'^archive/$', 'foo.views.blog_archive'),

)
```

In this example, the captured username variable is passed to the included URLconf and, hence, to *every* view function within that URLconf.

Note that the captured parameters will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those parameters as valid. For this reason, this technique is useful only if you're certain that every view in the included URLconf accepts the parameters you're passing.

How Extra URLconf Options Work with include()

Similarly, you can pass extra URLconf options to include(), just as you can pass extra URLconf options to a normal view – as a dictionary. When you do this, *each* line in the included URLconf will be passed the extra options.

For example, the following two URLconf sets are functionally identical.

Set one:

```
# urls.py


from django.conf.urls.defaults import *


urlpatterns = patterns('',
```

```
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

Set two:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *
```

```
urlpatterns = patterns('',

    (r'^archive/$', 'mysite.views.archive', {'blogid':
3}),

    (r'^about/$', 'mysite.views.about', {'blogid': 3}),

    (r'^rss/$', 'mysite.views.rss', {'blogid': 3}),

)
```

As is the case with captured parameters (explained in the previous section), extra options will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those options as valid. For this reason, this technique is useful only if you're certain that every view in the included URLconf accepts the extra options you're passing.