Server-side website programming (https://developer.mozilla.org/en-US/docs/Learn/Server-side)

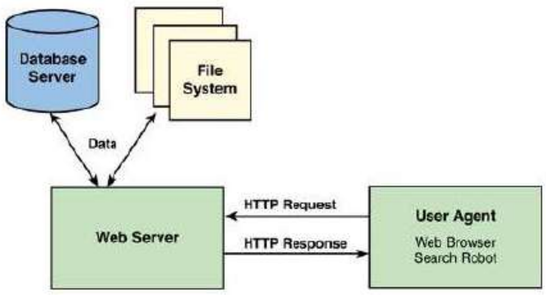
- The Dynamic Websites – **Server-side programming** topic is a series of modules that show how to create dynamic websites; websites that deliver customized information in response to HTTP requests.
- The modules provide a general introduction to server-side programming, along with specific beginner-level guides on how to use the Django (Python) and Express (Node.js/JavaScript) web frameworks to create basic applications.
- Most major websites use some kind of server-side technology to dynamically display data as required. For example, imagine how many products are available on Amazon, and imagine how many posts have been written on Facebook. Displaying all of these using different static pages would be extremely inefficient, so instead such sites display static templates (built using **HTML**, **CSS**, and **JavaScript**), and then dynamically update the data displayed inside those templates when needed, such as when you want to view a different product on Amazon.
- In the modern world of web development, learning about server-side development is highly recommended.

Learning pathway

- Expertise in client-side coding is not required, but a basic knowledge will help you work better with the developers creating your client-side web "front end".
- You will need to understand "how the web works". We recommend that you first read the following topics:
    - What is a web server
    - What software do I need to build a website?
    - How do you upload files to a web server?

What is a web server?

- The term web server can refer to **hardware** or **software**, or **both of them working together**.
- On the **hardware side**, a web server is a computer that stores web server software and a website's component files (for example, HTML documents, images, CSS stylesheets, and JavaScript files). A web server connects to the Internet and supports physical data interchange with other devices connected to the web.



- On the **software side**, a web server includes several parts that control how web users access hosted files. At a minimum, this is an **HTTP server**. An HTTP server is software that understands URLs (web addresses) and HTTP (the protocol your browser uses to view webpages). An HTTP server can be accessed through the domain names of the websites it stores, and it delivers the content of these hosted websites to the end user's device.
    - A **static web server**, or stack, consists of a computer (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files as-is to your browser.
    - A **dynamic web server** consists of a static web server plus extra software, most commonly an application server and a database. We call it "dynamic" because the application server updates the hosted files before sending content to your browser via the HTTP server.

Module-1: MVC based Web Designing

Web framework, MVC Design Pattern. Django Evolution, Views, Mapping URL to Views, Working of Django URL Confs and Loose Coupling, Errors in Django, Wild Card Patterns in URLs.

What Is a Web Framework?

A Web framework provides

- a programming infrastructure for your applications, so that you can focus on writing clean, maintainable code without having to reinvent the wheel. In a nutshell, that's what Django does.

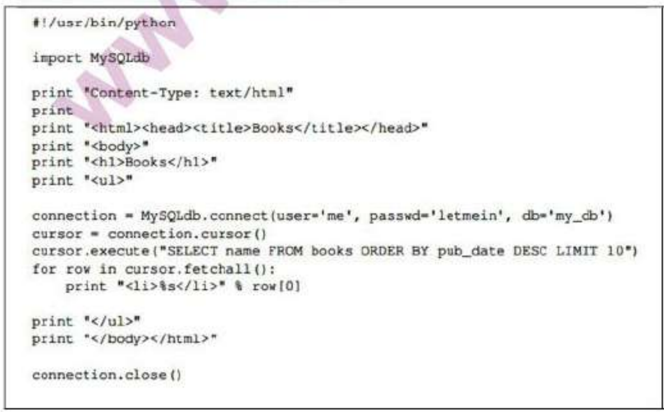
But as a Web application grows beyond the trivial, this approach breaks down, and you face a number of problems:

- **What happens when multiple pages need to connect to the database?** Surely that database-connecting code shouldn't be duplicated in each individual CGI script, so the pragmatic thing to do would be to refactor it into a shared function.
- **Should a developer really have to worry about printing the "Content-Type" line and remembering to close the database connection?** This sort of boilerplate reduces programmer productivity and introduces opportunities for mistakes. These setup- and teardown-related tasks would best be handled by some common infrastructure.
- **What happens when this code is reused in multiple environments, each with a separate database and password?** At this point, some environment-specific configuration becomes essential.
- **What happens when a Web designer who has no experience coding Python wishes to redesign the page?** Ideally, the logic of the page — the retrieval of books from the database — would be separate from the HTML display of the page, so that a designer could edit the latter without affecting the former.

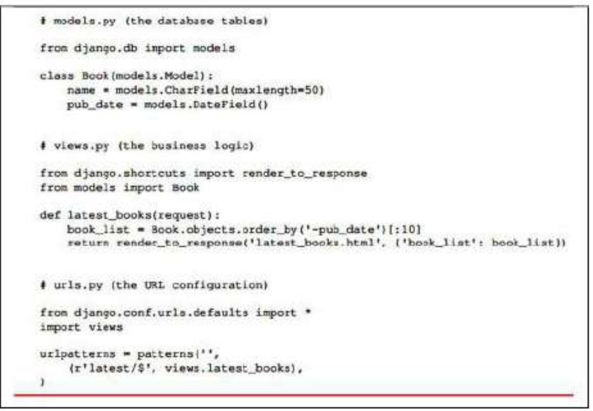
These problems are precisely what a Web framework intends to solve.

==

The MVC Design Pattern



```
#!/usr/bin/python

import MySQLdb

print "Content-Type: text/html"
print
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"

connection = MySQLdb.connect(user='me', passwd='letmein', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC LIMIT 10")
for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]

print "</ul>"
print "</body></html>"

connection.close()
```



```
# models.py (the database tables)

from django.db import models

class Book(models.Model):
    name = models.CharField(maxlength=50)
    pub_date = models.DateField()


# views.py (the business logic)

from django.shortcuts import render_to_response
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})


# urls.py (the URL configuration)

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

urlpatterns = patterns('',
    (r'latest/$', views.latest_books),
)
```

Prof. Parthasarathy P V (M) 8961623822

```
# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

The main thing to note here is the separation of concerns:

- The ***models.py*** file contains a description of the database table, as a Python class. This is called a ***model***. Using this class, you can create, retrieve, update, and delete records in your database using simple Python code rather than writing repetitive SQL statements.
- The ***views.py*** file contains the business logic for the page, in the ***latest_books()*** function. This function is called a ***view***.
- The ***urls.py*** file specifies which view is called for a given URL pattern. In this case, the URL ***/latest/*** will be handled by the ***latest_books()*** function.
- The ***latest_books.html*** is an HTML template that describes the design of the page.

Taken together, these pieces loosely follow the ***Model-View-Controller (MVC)*** design pattern.

- MVC defines a way of developing software so that the code for defining and accessing data ***(the model)*** is separate from request routing logic ***(the controller),*** which in turn is separate from the user interface ***(the view).***
- A **key advantage** of such an approach is that **components are loosely coupled**.
- That is, each distinct piece of a Django-powered Web application has a single key purpose and can be changed independently without affecting the other pieces.
- For example, a developer can change the URL for a given part of the application without affecting the underlying implementation.
- A designer can change a page's HTML without having to touch the Python code that renders it. A database administrator can rename a database table and specify the change in a single place, rather than having to search and replace through a dozen files.

===

Django's History

- If you've been building Web applications for a while, you're probably familiar with the problems in the CGI example we presented earlier. The classic Web developer's path goes something like this:
    1. Write a Web application from scratch.
    2. Write another Web application from scratch.
    3. Realize the application from step 1 shares much in common with the application from step 2.
    4. Refactor the code so that application 1 shares code with application 2.
    5. Repeat steps 2-4 several times.
    6. Realize you've invented a framework.

This is precisely how Django itself was created!

- Django grew organically from real-world applications written by a Web development team in **Lawrence, Kansas**. It was born in the **fall of 2003**, when the **Web programmers at the**

Lawrence Journal-World newspaper, Adrian Holovaty and Simon Willison, began using **Python to build applications.**

- The World Online team, responsible for the production and maintenance of several local news sites, thrived in a development environment dictated by journalism deadlines.
- For the sites — including **LJWorld.com**, **Lawrence.com, and KUsports.com — journalists** (and management) demanded that features be added and entire applications be built on an intensely fast schedule, often with only days' or hours' notice.
- Thus, Adrian and Simon developed a time-saving Web development framework out of necessity — it was the only way they could build maintainable applications under the extreme deadlines.
- **In summer 2005**, after having developed this framework to a point where it was efficiently powering most of World Online's sites, the World Online team, which now included Jacob Kaplan-Moss, decided to release the framework as open source software. They released it in **July 2005 and named it Django, after the jazz guitarist Django Reinhardt.**
- Although Django is now an open source project with contributors across the planet, the original World Online developers still provide central guidance for the framework's growth, and World Online contributes other important aspects such as employee time, marketing materials, and hosting/bandwidth for the framework's Web site (http://www.djangoproject.com/).
- This history is relevant because it helps explain two key matters.
    1. The first is Django's "sweet spot." Because Django was born in a news environment, it offers **several features (particularly its admin interface) that are particularly well suited for "content" sites — sites like eBay, craigslist.org, and washingtonpost.com** that **offer dynamic, database-driven information.**
    2. The second matter to note is how Django's origins have **shaped the culture of its open source community**. Because Django was extracted from real-world code, rather than being an academic exercise or commercial product, it is acutely focused on solving Web development problems that Django's developers themselves have faced — and continue to face.

Required Programming Knowledge

- Readers of this book should understand the basics of procedural and **object-oriented programming**: **control structures** (if, while, and for), **data structures** (lists, hashes/dictionaries), **variables, classes, and objects**.
- Experience in Web development is, as you may expect, very helpful, but it's not required to read this book.

Required Python Knowledge

- At its core, Django is simply a collection of libraries written in the Python programming language. To develop a site using Django, you write Python code that uses these libraries.
- Learning Django, then, is a matter of learning how to program in Python and understanding how the Django libraries work.
- If you have experience programming in Python, you should have no trouble diving in.
- By and large, the Django code doesn't perform "black magic" (i.e., programming trickery whose implementation is difficult to explain or understand). For you, learning Django will be a matter of learning Django's conventions and APIs.

- Python tutorial, available online at http://docs.python.org/tut/. We also recommend Mark Pilgrim's free book Dive Into Python, available at http://www.diveintopython.org/ and published in print by Apress.

==

Django - Views

- As our first goal, let's create a Web page that displays the current date and time. This is a good example of a *dynamic* Web page, because the contents of the page are not static.
- To create this page, we'll write a ***view*** function. A ***view*** function, or ***view*** for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really.
- The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path.
- Let's create a file called ***views***.py in the ***mysite*** directory.
- Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime
def current_datetime(request):
        now = datetime.datetime.now()
        html = "<html><body>It is now %s.</body></html>" % now
        return HttpResponse(html)
```

- The first line of code within the function calculates the current date/time as a **datetime. datetime** object, and stores that as the local variable now.
- The second line of code within the function constructs an HTML response using Python's format-string capability. The **%s** within the string is a placeholder, and the percent sign after the string means "Replace the %s with the value of the variable now."
- Finally, the view returns an **HttpResponse** object that contains the generated response. Each view function is responsible for returning an **HttpResponse** object.

==

Mapping URL to Views

- This view function returns an HTML page that includes the current date and time. But how do we tell Django to use this code? That's where ***URLconfs*** come in.
- A ***URLconf*** is ***like a table of contents for your Django-powered Web site***.
- Basically, it's a **mapping between URL patterns and the view functions** that should be called for those URL patterns. It's how you tell Django, "For this URL, call this code".
- Remember that the **view functions need to be on the Python path**.
- When you executed **django-admin.py startproject**, the script created a **URLconf** for us automatically: the file ***urls.py***. By default, it looks something like this--------→

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

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.apps.foo.urls.foo')),

    # Uncomment this for admin:
#     (r'^admin/', include('django.contrib.admin.urls')),
)
```

- Let's step through this code one line at a time:
    - The first line imports all objects from the ***django.conf.urls.defaults*** module, including a function called ***patterns***.
    - The second line calls the function ***patterns()*** and ***saves the result*** into a variable called ***urlpatterns***. The ***patterns()*** function gets passed only a single argument—the empty string.
- The main thing to note here is the variable ***urlpatterns***, which Django expects to find in your **ROOT_URLCONF** module. This variable defines the **mapping between URLs and the code that handles those URLs**.
- By default, everything in the ***URLconf*** is commented out—your Django application is a **blank slate**.

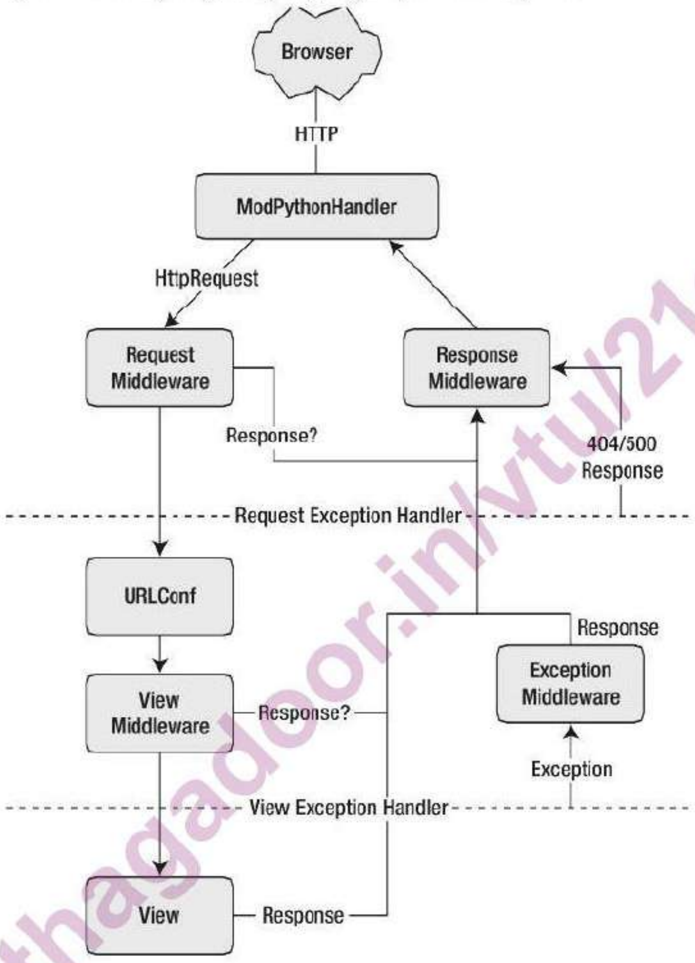
- Let's edit this file to expose our ***current_datetime*** view:

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

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
)
```

- We made two changes here.
    - First, we imported the current_datetime view from its module (mysite/views.py, which translates into mysite.views in Python import syntax).
    - Next, we added the line (r'^time/$', current_datetime),. This line is referred to as a URLpattern—it's a Python tuple in which the first element is a simple regular expression and the second element is the view function to use for that pattern.
- In a nutshell, we just told Django that any request to the URL **/time/** should be handled by the **current_datetime** view function. A few things are worth pointing out:
    - Note that, in this example, we passed the current_datetime view function as an object without calling the function. This is a key feature of Python (and other dynamic languages): functions are first-class objects, which means you can pass them around just like any other variables.
    - The **r** in **r'^time/$'** means that **'^time/$** is a Python raw string. This allows regular expressions to be written without overly verbose escaping.
    - We should exclude the expected slash at the beginning of the **'^time/$'** expression in order to match **/time/**. ***Django automatically puts a slash before every expression***.
    - At first glance, this may seem odd, but **URLconfs** can be included in other **URLconfs**, and leaving off the leading slash simplifies matters.
    - The **caret character (^)** and **dollar sign character ($)** are important.
        - The **caret** means **"require that the pattern matches the start of the string,"**
        - The **dollar** sign means **"require that the pattern matches the end of the string."**

How Django Processes a Request: Complete Details

Figure. *The complete flow of a Django request and response*



==

Working of Django URLConfs and Loose Coupling

- Now's a good time to highlight a key ***philosophy behind URLconfs*** and ***behind Django*** in general: the principle of ***loose coupling***.
- ***Loose coupling*** is a software-development approach that values the importance of making pieces interchangeable. If two pieces of code are loosely coupled, then changes made to one of the pieces will have little or no effect on the other.
- Django's ***URLconfs*** are a good example of this principle in practice.
- In a Django Web application, the ***URL definitions and the view functions*** they call are ***loosely coupled***; that is, the decision of what the URL should be for a given function, and the implementation of the function itself, reside in two separate places. This lets a developer switch out one piece without affecting the other.
- In contrast, ***other Web development platforms couple the URL to the program***. In ***typical PHP*** (http://www.php.net/) applications, for example, the URL of your application is designated by where you place the code on your ***filesystem***. In early versions of the CherryPy Python Web framework (http://www.cherrypy.org/), the URL of your application

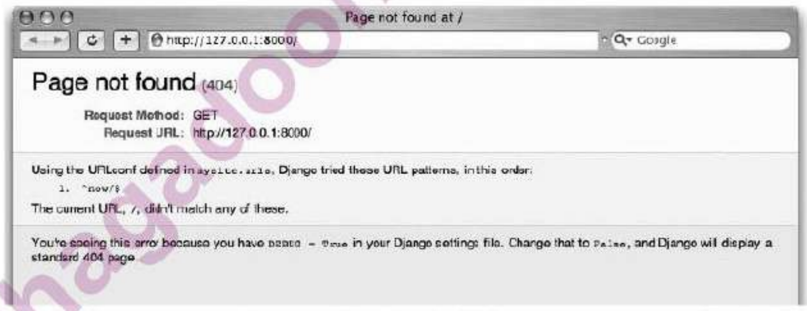
corresponded to the name of the method in which your code lived. This may seem like a convenient shortcut in the short term, but it can get unmanageable in the long run.

- For example, consider the view function we wrote earlier, which displays the current date and time. If we wanted to change the URL for the application— say, move it from ***/time/*** to ***/currenttime/***—we could make a quick change to the ***URLconf***, without having to worry about the underlying implementation of the function.
- Similarly, if we ***wanted to change the view function***—***altering its logic somehow***—we could do that ***without affecting the URL to which the function is bound***.
- Furthermore, if we ***wanted to expose the current-date functionality at several URLs,*** we could easily take care of that by ***editing the URLconf***, without having to touch the view code. That's loose coupling in action.

==

404 Errors in Django

- In our URLconf thus far, we've defined only a single ***URLpattern***: the one that handles requests to the URL ***/time/***. What happens when a different URL is requested?
- To find out, try running the Django development server and hitting a page such as http://127.0.0.1:8000/hello/ or http://127.0.0.1:8000/does-not-exist/, or even http://127.0.0.1:8000/ (the site "root"). You should see a "Page not found" message (see the below Figure).
- Django displays this message because you requested a ***URL that's not defined in your URLconf***.


Page not found at /

http://127.0.0.1:8000/

Page not found (404)

Request Method: GET
Request URL: http://127.0.0.1:8000/

Using the URLconf defined in mysite.urls, Django tried these URL patterns, in this order:

1. ^now/$

The current URL, /, didn't match any of these.

You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.

- The utility of this page goes beyond the basic 404 error message; it also tells you precisely which URLconf Django used and every pattern in that URLconf. From that information, you should be able to tell why the requested URL threw a 404.
- Naturally, this is sensitive information intended only for you, the Web developer. If this were a production site deployed live on the Internet, we wouldn't want to expose that information to the public. For that reason, this "Page not found" page is only displayed if your Django project is in *debug mode*. We'll explain how to deactivate debug mode later.
- For now, just know that every Django project is in debug mode when you first create it, and if the project is not in debug mode, a different response is given.

Django's Pretty Error Pages

- Take a moment to admire the fine Web application we've made so far - deliberately introduce a Python error into our ***views.py*** file by commenting out the ***offset = int(offset)*** line in the ***hours_ahead*** view:

```
def hours_ahead(request, offset):
    #offset = int(offset)
```

```
dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
return HttpResponse(html)
```

- Load up the development server and navigate to ***/time/plus/3/.*** You'll see an error page with a significant amount of information, including a ***TypeError*** message displayed at the very top: "***unsupported type for timedelta hours component: str***".
- What happened? The ***datetime.timedelta*** function expects the ***hours*** parameter to
- be an ***integer***, and we commented out the bit of code that converted ***offset*** to an integer. That caused ***datetime.timedelta*** to raise the ***TypeError***. It's the typical kind of small bug that every programmer runs into at some point.

The point of this example was to demonstrate Django's error pages. Take some time to explore the error page and get to know the various bits of information it gives you.

- The Django error page is capable of displaying more information in certain special cases, such as the case of template syntax errors.
- Are you the type of programmer who likes to debug with the help of carefully placed print statements? You can use the Django error page to do so—just without the print statements. At any point in your view, temporarily insert an assert False to trigger the error page. Then, you can view the local variables and state of the program.
- Finally, it's obvious that much of this information is sensitive—it exposes the innards of your Python code and Django configuration—and it would be foolish to show this information on the public Internet. A malicious person could use it to attempt to reverse-engineer your Web application and do nasty things. For that reason, the Django error page is only displayed when your Django project is in debug mode.

==

Dynamic URLs

- In our ***first view example***, the contents of the page—the ***current date/time—were dynamic***, but the ***URL (/time/) was static***.
- In most dynamic Web applications, though, a URL contains parameters that influence the output of the page.
- Let's ***create a second view that displays the current date and time offset by a certain number of hours***. The goal is to craft a site in such a way that the page /time/plus/1/ displays the date/time one hour into the future, the page /time/plus/2/ displays the date/time two hours into the future, the page /time/plus/3/ displays the date/time three hours into the future, and so on.
- A novice might think to code a separate view function for each hour offset, which might result in a ***URLconf*** like this:

```
urlpatterns = patterns('',
(r'^time/$', current_datetime),
(r'^time/plus/1/$', one_hour_ahead),
(r'^time/plus/2/$', two_hours_ahead),
(r'^time/plus/3/$', three_hours_ahead),
(r'^time/plus/4//$', four_hours_ahead),
)
```

Pretty URLs

- If you're experienced in another Web development platform, such as PHP or Java, you may be thinking, "Hey, let's use a query string parameter!"—something like /time/plus?hours=3, in which the hours would be designated by the hours parameter in the URL's query string (the part after the ?).
- You can do that with Django, but one of Django's core philosophies is that URLs should be beautiful. The URL /time/plus/3/ is far cleaner, simpler, more readable, easier to recite to somebody aloud and . . . just plain prettier than its query string counterpart.
- ***Pretty URLs are a sign of a quality Web application***. Django's URLconf system encourages pretty URLs by making it easier to use pretty URLs than not to.

Wild Card patterns in URLS

- Continuing with our ***hours_ahead*** example, let's put a wildcard in the ***URLpattern***. As we mentioned previously, a URLpattern is a regular expression; hence, we can use the regular expressionpattern **\d+** to match one or more digits:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead
urlpatterns = patterns('',
(r'^time/$', current_datetime),
(r'^time/plus/\d+/$', hours_ahead),
)
```

- This URLpattern will match any URL such as /time/plus/2/, /time/plus/25/, or even /time/plus/100000000000/. Come to think of it, let's limit it so that the maximum allowed offset is 99 hours. That means we want to allow either one- or two-digit numbers—in regular expression syntax, that translates into \d{1,2}:

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

- Now that we've designated a wildcard for the URL, we need a way of passing that data to the view function, so that we can use a single view function for any arbitrary hour offset. We do this by placing parentheses around the data in the URLpattern that we want to save.
- In the case of our example, we want to save whatever number was entered in the URL, so let's put parentheses around the \d{1,2}:

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

- The final URLconf, including our previous current_datetime view, looks like this:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

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

- ***hours_ahead*** is very similar to the ***current_datetime*** view we wrote earlier, with a key difference: it takes an extra argument, the number of hours of offset. Add this to views.py:

```
def hours_ahead(request, offset):
        offset = int(offset)
        dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
        html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
        return HttpResponse(html)
```

Let's step through this code one line at a time:

- Just as we did for our current_datetime view, we import the class django.http. HttpResponse and the datetime module.
- The view function, hours_ahead, takes two parameters: request and offset.
- request is an HttpRequest object, just as in current_datetime. We'll say it again: each view always takes an HttpRequest object as its first parameter.
- offset is the string captured by the parentheses in the URLpattern. For example, if the requested URL were /time/plus/3/, then offset would be the string '3'. If the requested URL were /time/plus/21/, then offset would be the string '21'. Note that captured strings will always be strings, not integers, even if the string is composed of only digits, such as '21'.
- The first thing we do within the function is call int() on offset. This converts the string value to an integer.
- The next line of the function shows why we called int() on offset. On this line, we calculate the current time plus a time offset of offset hours, storing the result in dt. The datetime.timedelta function requires the hours parameter to be an integer.
- Next, we construct the HTML output of this view function, just as we did in current_ datetime. A small difference in this line from the previous line is that it uses Python's format-string capability with two values, not just one. Hence, there are two %s symbols in the string and a tuple of values to insert: (offset, dt).
- Finally, we return an HttpResponse of the HTML—again, just as we did in current_datetime.

With that view function and URLconf written, start the Django development server, and visit http://127.0.0.1:8000/time/plus/3/ to verify it works. Then try http://127.0.0.1:8000/time/plus/5/. Then http://127.0.0.1:8000/time/plus/24/. Finally, visit http://127.0.0.1:8000/time/plus/100/ to verify that the pattern in your URLconf only accepts one- or two-digit numbers; Django should display a "Page not found" error in this case, just as we saw in the "404 Errors" section earlier. The URL http://127.0.0.1:8000/time/plus/ (with no hour designation) should also throw a 404.

- If you're following along while coding at the same time, you'll notice that the views.py file now contains two views. ***views.py*** should look like this:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
        now = datetime.datetime.now()
        html = "<html><body>It is now %s.</body></html>" % now
        return HttpResponse(html)

def hours_ahead(request, offset):
        offset = int(offset)
        dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
        html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
        return HttpResponse(html)
```

==