HIGH TECH

ДЖЕФФ ФОРСЬЕ
ПОЛ БИССЕКС
УЭСЛИ ЧАН

Django

РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ НА PYTHON

СИМВОЛ®

Серия «High tech»

Джефф Форсье, Пол Биссекс, Уэсли Чан

Django. Разработка веб-приложений на Python

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Форсье Дж., Биссекс П., Чан У.
Django. Разработка веб-приложений на Python. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 456 с., ил.
ISBN: 978-5-93286-167-7

На основе простой и надежной платформы Django на Python можно создавать мощные веб-решения всего из нескольких строк кода. Авторы, опытные разработчики, описывают все приемы, инструменты и концепции, которые необходимо знать, чтобы оптимально использовать Django 1.0, включая все основные особенности новой версии.

Это полное руководство начинается с введения в Python, затем подробно обсуждаются основные компоненты Django (модели, представления и шаблоны) и порядок организации взаимодействия между ними. Описываются методы разработки конкретных приложений: блог, фотогалерея, система управления содержимым, инструмент публикации фрагментов кода с подсветкой синтаксиса. После этого рассматриваются более сложные темы: расширение системы шаблонов, синдицирование, настройка приложения администрирования и тестирование веб-приложений.

Авторы раскрывают разработчику секреты Django, давая подробные разъяснения и предоставляя большое количество примеров программного кода, сопровождая их построчным описанием и иллюстрациями.

ISBN: 978-5-93286-167-7
ISBN: 978-0-13-235613-8 (англ)

© Издательство Символ-Плюс, 2009
Authorized translation of the English edition © 2009 Pearson Education, Inc. This translation is published and sold by permission of Pearson Education, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 17.12.2009. Формат 70x100 1/16. Печать офсетная.
Объем 28,5 печ. л. Доп. тираж 1000 экз. Заказ № 1496
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Брайану Левайну (Brian Levine), открывшему мне Python – маленький шаг, имевший большое значение. Моим родителям – за то, что позволили мне монополизировать домашний компьютер, пока я рос. И моей супруге, за ее любовь, поддержку и понимание.

Джефф Форсье

Моему покойному отцу Генри, научившему меня работать; моей матери Гленде, научившей меня писать; и моей жене Кетлин, озарившей мою жизнь.

Пол Биссекс

Моим замечательным детям Леанне Син-Йи и Дейлину Син-Жи, которые вынудили меня выработать способность находиться во всех местах одновременно, чтобы уследить за ними, и которые чудесным образом способны перемещать меня назад во времени, напоминая мне мое детство с его удивительными чудесами.

Уэсли Чан

Оглавление

Предисловие 17

Благодарности 23

Введение 26

I. Введение 31

1. Практическое введение в Python для Django 33

Практические приемы владения Python и Django 34
Введение: интерактивный интерпретатор языка Python 35
Основы Python 37
Комментарии 37
Переменные и присваивание значений 38
Операторы 38
Стандартные типы данных в языке Python 39
Логические значения объектов 39
Числа 40
Числовые операторы 41
Встроенные и фабричные функции для работы с числами 42
Последовательности и итерируемые объекты 43
Списки 46
Строки 49
Встроенные и фабричные функции последовательностей 56
Отображения: словари 57
В заключение о стандартных типах данных 60
Управление потоком выполнения 60
Условная инструкция 61
Циклы 61
Обработка исключений 63
Предложение finally 64
Возбуждение исключений с помощью инструкции raise 64
Файлы 66

Функции . . . 67
Объявление и вызов функций . . . 67
Функции – это обычные объекты . . . 70
Анонимные функции . . . 72
*args и **kwargs . . . 74
Декораторы . . . 78
Объектно-ориентированное программирование . . . 79
Определение классов . . . 80
Создание экземпляров . . . 81
Создание подклассов . . . 82
Вложенные классы . . . 83
Регулярные выражения . . . 84
Модуль re . . . 84
Поиск и соответствие . . . 84
Типичные ошибки . . . 85
Кортежи с единственным элементом . . . 85
Модули . . . 85
Изменяемость . . . 87
Конструктор и метод инициализации . . . 90
Стиль оформления программного кода (PEP 8 и не только) . . . 91
Отступы в четыре пробела . . . 91
Используйте пробелы, но не символы табуляции . . . 92
Не записывайте короткие блоки программного кода в одной строке с заголовком инструкции . . . 92
Создавайте строки документирования . . . 92
В заключение . . . 94

2. Django для нетерпеливых: создание блога . . . 95

Создание проекта . . . 96
Запуск сервера разработки . . . 98
Создание приложения блога . . . 100
Создание модели . . . 101
Настройка базы данных . . . 101
Использование сервера баз данных . . . 102
Использование SQLite . . . 102
Создание таблиц . . . 104
Настройка автоматизированного приложения администрирования . . . 105
Опробование приложения администрирования . . . 107
Создание общедоступного раздела приложения блога . . . 112
Создание шаблона . . . 112
Создание функции представления . . . 113
Создание шаблона адреса URL . . . 114

Заключительные штрихи 115
Усовершенствование шаблона 115
Упорядочение по дате 116
Форматирование даты и времени с помощью фильтра 118
В заключение 118

3. Начало 120

Основы динамических веб-сайтов 121
Взаимодействие: HTTP, URL, запросы, ответы 121
Хранилища данных: SQL и реляционные базы данных 121
Представление: шаблоны отображения в разметку HTML и в другие форматы 122
Сложим все вместе 123
Понимание моделей, представлений и шаблонов 123
Выделение уровней (MVC) 123
Модели 125
Представления 125
Шаблоны 126
Общий обзор архитектуры Django 126
Основные принципы Django 128
Django стремится быть Питонической 128
Не повторяйся (Don't Repeat Yourself, DRY) 129
Слабая зависимость и гибкость 129
Быстрая разработка 130
В заключение 131

II. Подробно о Django 133

4. Определение и использование моделей 135

Определение моделей 135
Преимущества ORM 135
Богатый набор типов полей в Django 137
Отношения между моделями 140
Наследование модели 145
Вложенный класс Meta 149
Регистрация в приложении администрирования и дополнительные параметры 151
Использование моделей 152
Создание и изменение базы данных с помощью утилиты manage.py 152
Синтаксис запросов 154
Использование возможностей SQL, не предоставляемых платформой Django 164
В заключение 168

5. Адреса URL, механизмы HTTP и представления 170
Адреса URL 170
Введение в URLconf 171
Замещение кортежей функциями url 172
Использование нескольких объектов patterns 173
Включение других файлов URLconf с помощью функции include 174
Объекты функций и строки с именами функций 175
Моделирование HTTP: запросы, ответы и промежуточная обработка 176
Объекты запросов 177
Объекты ответов 180
Промежуточная обработка 181
Представления/управляющая логика 183
Просто функции на языке Python 183
Универсальные представления 184
Полууниверсальные представления 187
Собственные представления 188
В заключение 190

6. Шаблоны и обработка форм 191
Шаблоны 191
Понимание контекста 192
Синтаксис языка шаблонов 193
Формы 199
Определение форм 199
Заполнение форм 205
Проверка и очистка 207
Отображение форм 209
Виджеты 211
В заключение 214

III. Приложения Django в примерах 215

7. Фотогалерея 217
Модель 218
Подготовка к выгрузке файлов 219
Установка PIL 221
Проверка поля ImageField 222
Создание нашего собственного поля файла 224
Инициализация 225
Добавление атрибутов в поле 227
Сохранение и удаление миниатюры 228

Использование ThumbnailImageField 229
Применение принципа «не повторяйся» к адресам URL 231
Схема адресов элементов Item приложения 233
Соединяем все это с шаблонами 235
В заключение 240

8. Система управления содержимым 242

Что такое система управления содержимым? 242
Альтернатива системе управления содержимым: Flatpages 243
Включение приложения Flatpages 244
Шаблоны Flatpages 245
Тестирование 246
За рамками Flatpage: простая система управления содержимым 247
Создание модели 248
Импортирование 250
Заключительная модель 250
Управление доступностью статей для просмотра 251
Работа с форматом Markdown 252
Шаблоны адресов URL в urls.py 255
Представления административного раздела 257
Отображение содержимого с помощью универсальных представлений 260
Шаблоны 261
Отображение статей 263
Добавление функции поиска 264
Управление пользователями 267
Поддержка производственного процесса 268
Возможные улучшения 268
В заключение 270

9. Живой блог 271

Что такое Ajax? 272
В чем состоит польза Ajax 272
Проектирование приложения 273
Выбор библиотеки Ajax 274
Структура каталогов приложения 275
Внедрение технологии Ajax 279
Основы 279
Символ «X» в аббревиатуре Ajax (или XML и JSON) 280
Установка библиотеки JavaScript 281
Настройка и тестирование библиотеки jQuery 282
Создание функции представления 284

Использование функции представления в JavaScript 286
В заключение 288

10. Pastebin 290

Определение модели 291
Создание шаблонов 293
Определение адресов URL 294
Запуск приложения 296
Ограничение числа записей в списке последних поступлений 300
Подсветка синтаксиса 301
Удаление устаревших записей с помощью задания cron 302
В заключение 304

IV. Дополнительные возможности и особенности Django 305

11. Передовые приемы программирования в Django 307

Настройка приложения администрирования 307
Изменение расположения и стилей элементов с помощью параметра fieldsets 308
Расширение базовых шаблонов 310
Добавление новых представлений 312
Декораторы аутентификации 312
Приложение Syndication 314
Класс Feed 314
Определение адреса URL ленты 316
Дополнительные возможности работы с лентами 317
Создание загружаемых файлов 317
Конфигурационные файлы Nagios 318
vCard 319
Значения, разделенные запятыми (CSV) 320
Вывод диаграмм и графиков с помощью библиотеки PyCha 321
Расширение механизма ORM с помощью собственных подклассов Manager 323
Изменение множества объектов, возвращаемых по умолчанию 324
Добавление новых методов в подклассы Manager 325
Расширение системы шаблонов 326
Простые специализированные теги шаблонов 326
Теги включения 330
Специализированные фильтры 333
Более сложные специализированные теги шаблонов 336
Альтернативные системы шаблонов 336
В заключение 338

12. Передовые приемы развертывания Django 339
Создание вспомогательных сценариев 339
Задания cron, выполняющие очистку 340
Импорт/экспорт данных 341
Изменение программного кода самой платформы Django 343
Кэширование 343
Типичный пример кэширования 344
Стратегии кэширования 347
Типы механизмов кэширования 352
Тестирование приложений на платформе Django 356
Основы доктестов 357
Основы модульного тестирования 358
Запуск тестов 358
Тестирование моделей 359
Тестирование всего веб-приложения в целом 361
Тестирование программного кода самой платформы Django 362
В заключение 364

V. Приложения 365

A. Основы командной строки 367
Ввод «команды» в «командную строку» 368
Ключи и аргументы 371
Каналы и перенаправление 373
Переменные окружения 375
Пути 377
В заключение 379

B. Установка и запуск Django 380
Python 380
Mac OS X 381
UNIX/Linux 381
Windows 381
Обновление путей поиска 382
Тестирование 384
Необязательные дополнения 386
Django 388
Официальные выпуски 388
Версия в разработке 388
Установка 388
Тестирование 389
Веб-сервер 389

Встроенный сервер: не для работы в нормальном режиме эксплуатации . . . 390
Стандартный подход: Apache и mod_python . . . 390
Гибкая альтернатива: WSGI . . . 394
Другой подход: flup и FastCGI . . . 395
База данных SQL . . . 396
SQLite . . . 396
PostgreSQL . . . 397
MySQL . . . 398
Oracle . . . 400
Прочие базы данных . . . 400
В заключение . . . 401

C. Инструменты разработки для платформы Django . . . 402

Управление версиями . . . 402
Ствол и ветви . . . 403
Слияние . . . 404
Централизованное управление версиями . . . 404
Децентрализованное управление версиями . . . 405
Управление версиями в вашем проекте . . . 406
Программное обеспечение управления проектами . . . 409
Trac . . . 409
Текстовые редакторы . . . 410
Emacs . . . 410
Vim . . . 411
TextMate . . . 411
Eclipse . . . 411

D. Поиск, оценка и использование приложений на платформе Django . . . 412

Где искать приложения . . . 413
Как оценивать приложения . . . 413
Как пользоваться приложениями . . . 414
Передача собственных приложений . . . 415

E. Django и Google App Engine . . . 416

Назначение платформы App Engine . . . 417
Приложения, опирающиеся исключительно на использование App Engine . . . 417
Ограничения платформы App Engine . . . 418
Проект Google App Engine Helper для Django . . . 418
Получение SDK и Helper . . . 419
Подробнее о Helper . . . 419

Интегрирование App Engine 420
Копирование программного кода App Engine в проект 420
Интегрирование App Engine Helper 421
Перенос приложения на платформу App Engine 422
Опробование 423
Добавление данных 424
Создание нового приложения на платформе Django, использующего возможности App Engine 425
В заключение 426
Ресурсы в Интернете 427

F. Участие в проекте Django 428

Алфавитный указатель 430

Предисловие

Добро пожаловать в Django!

Поздравляем вас и добро пожаловать в Django! Мы рады, что вы присоединились к нашему путешествию. Вы откроете для себя мощную платформу разработки веб-приложений, которая позволит вам быстро выполнять свою работу, – от проектирования и разработки оригинального приложения до его обновления и расширения его возможностей без необходимости вносить существенные изменения в программный код.

Об этой книге

В магазинах уже можно найти несколько книг о Django, но наша книга отличается от них тем, что основное внимание в ней уделяется трем областям: изучению основ Django, различных примеров приложений и дополнительных свойств Django. Наша цель состоит в том, чтобы в этой книге настолько полно охватить предмет обсуждения, чтобы вы нашли ее полезной независимо от уровня своей подготовленности и получили полное представление о платформе и ее возможностях.

Путеводитель по главам

На рис. 1 вы увидите рекомендуемые стартовые точки начала чтения книги в зависимости от вашего уровня знания языка программирования Python и платформы Django. Безусловно, мы рекомендуем прочитать ее от корки до корки, и тем не менее данная диаграмма поможет вам, если вы ограничены во времени. В любой момент независимо от уровня подготовки вы можете перейти к изучению приложений, потому что чтение и изучение программного кода – один из лучших способов обучения. Ниже приводится краткое содержание по главам, чтобы помочь вам выяснить, что следует прочитать.

Часть I «Введение»

Часть I содержит базовые сведения, необходимые для тех, кто не знаком еще с Django и/или Python, хотя даже опытным читателям мы рекомендуем в чтении начать с главы 3 «Начало».

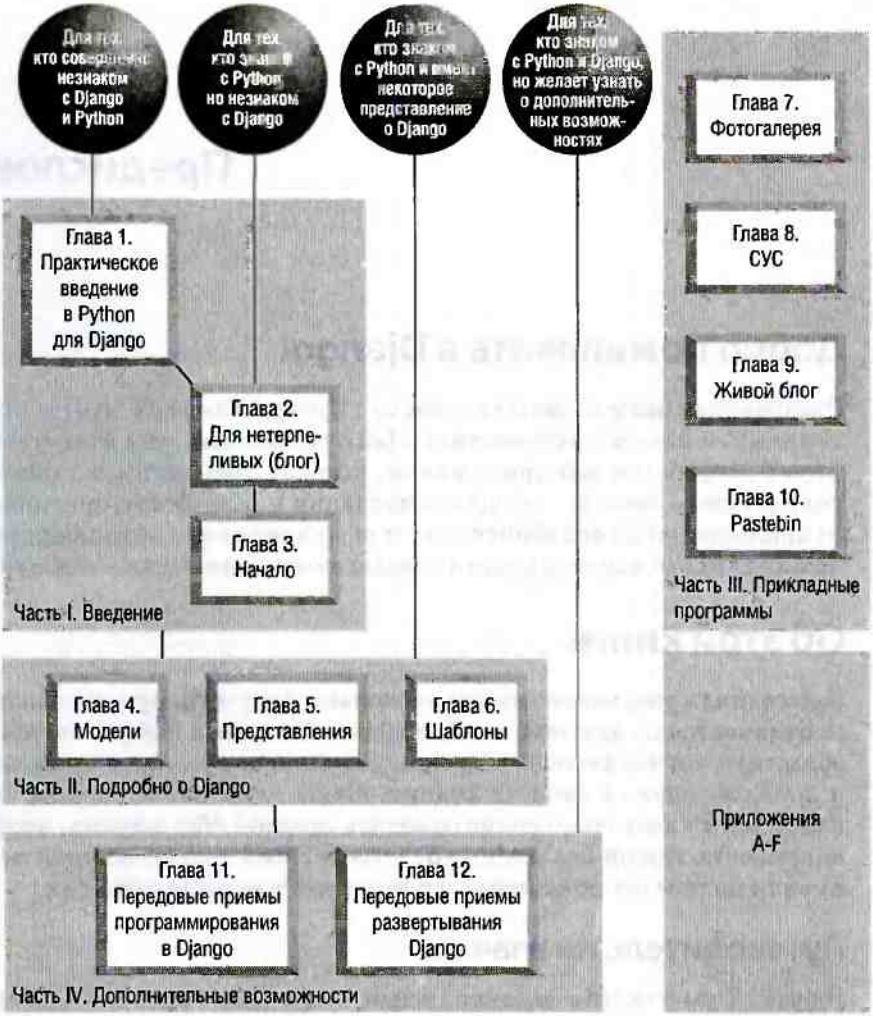

Рис. 1. *Рекомендуемые отправные точки для чтения в зависимости от уровня подготовки*

Глава 1 «Практическое введение в Python для Django»

Эта вводная глава предназначена для тех, кто не знаком с языком программирования Python. В этой главе демонстрируются не только синтаксические особенности, но также углубленный взгляд на модель распределения памяти в Python и типы данных, особенно на те, что наиболее часто используются в Django.

Глава 2 «Django для нетерпеливых: создание блога»

Эта глава предназначена для тех, кто, перепрыгнув через введение в язык программирования Python, стремится сразу же погрузиться в изучение приложения, которое можно создать на платформе Django за 15–20 минут. Эта глава содержит отличный обзор возможностей платформы Django.

Глава 3 «Начало»

Эта глава предназначена для более методичных читателей и служит введением в основы разработки веб-приложений (в ней содержатся полезные сведения как для начинающих, так и для опытных программистов). После введения всех формальностей здесь описывается место и роль каждой концепции в мире Django, а также философия платформы и ее отличия от других платформ.

Часть II «Подробно о Django»

Вторая часть описывает базовые компоненты платформы, закладывая фундамент для примеров приложений в третьей части «Приложения Django в примерах»

Глава 4 «Определение и использование моделей»

В главе рассказывается, как определять и использовать модели данных, включая основную объектно-реляционную проекцию (Object-Relational Mapping, ORM) Django, от простых полей до сложных схем отношений.

Глава 5 «Адреса URL, механизмы HTTP и представления»

В этой главе подробно описывается, как платформа Django работает с адресами URL и остальными особенностями протокола HTTP, включая промежуточную обработку, а также как использовать универсальные представления Django и как создавать собственные представления или модернизировать существующие.

Глава 6 «Шаблоны и обработка форм»

Глава охватывает последнюю значительную часть платформы. Здесь исследуется язык шаблонов, используемый в Django, и механизмы обработки форм, а также рассказывается, как отобразить данные перед пользователями и как получать данные от них.

Часть III «Приложения Django в примерах»

В третьей части будут созданы четыре различных приложения, чтобы обозначить различные аспекты или компоненты, используемые в процессе разработки приложений на платформе Django. В каждом из примеров будут вводиться новые идеи и расширяться концепции, представленные в частях I и II.

Глава 7 «Фотогалерея»

В главе демонстрируется применение правила «не повторяй самого себя» к структуре адресов URL, а также создание поля с эскизами изображений для простого приложения фотогалереи.

Глава 8 «Система управления содержимым»

В главе рассказывается о двух взаимосвязанных подходах к созданию СУС и подобных им систем, а также охватывает вопросы использования некоторых приложений, «пожертвованных» проекту Django.

Глава 9 «Живой блог»

В главе демонстрируется процесс создания «живого блога» – сайта, написанного с использованием передовых приемов программирования на JavaScript, который может служить примером применения технологии AJAX в проектах на платформе Django и показывает, насколько просто можно использовать любые инструменты поддержки AJAX.

Глава 10 «Pastebin»

В главе демонстрируется широта возможностей универсальных представлений Django на примере веб-приложения pastebin, при создании которого практически не потребовалась наша собственная реализация логики.

Часть IV «Дополнительные возможности и особенности Django»

Часть IV представляет собой коллекцию более сложных тем, начиная от настройки приложения администратора Django и заканчивая созданием сценариев командной строки, способных взаимодействовать с приложениями на платформе Django.

Глава 11 «Передовые приемы программирования в Django»

Глава затрагивает темы, связанные с процессом разработки программного кода приложений, такие как генерирование лент RSS, расширение языка шаблонов и улучшение приложения администрирования Django.

Глава 12 «Передовые приемы развертывания Django»

В главе рассматривается множество хитростей, связанных с развертыванием приложений на платформе Django, и работа с приложением из-за пределов Django, например из сценариев командной строки, заданий планировщика cron, а также при тестировании и импортировании данных.

Часть V «Приложения»

Эта часть восполняет недостающую информацию и рассматривает остальные темы, которые не были выделены в отдельные главы. Здесь изучаются основы командной строки UNIX, проблемы установки

и стратегии развертывания Django, инструменты разработки и многое другое.

Приложение A «Основы командной строки»

В приложении рассматриваются основы командной строки UNIX для тех, кто раньше не сталкивался с ней. Поверьте – это очень полезно!

Приложение B «Установка и запуск Django»

В приложении демонстрируется процесс установки компонентов, необходимых для запуска Django, включая особенности установки при наличии различных баз данных и веб-серверов, а также даются некоторые советы по развертыванию.

Приложение C «Инструменты разработки для платформы Django»

В приложении перечисляются некоторые инструменты разработки, с которыми вы можете уже быть знакомы, включая управление версиями исходных текстов, текстовые редакторы и многое другое.

Приложение D «Поиск, оценка и использование приложений на платформе Django»

Хороший разработчик пишет свой программный код, но отличный разработчик использует повторно уже написанный кем-то ранее программный код! В приложении D мы поделимся некоторыми советами о том, где и как искать приложения Django для повторного использования.

Приложение E «Django и Google App Engine»

В приложении E рассматривается, как в приложениях на платформе Django можно использовать преимущества нового механизма Google App Engine и демонстрируется, как запускать приложения на платформе Django под управлением платформы App Engine.

Приложение F «Участие в проекте Django»

В приложении вы узнаете, как принять участие в развитии проекта Django и как стать членом сообщества.

Типографские соглашения

На протяжении всей книги **жирным шрифтом** будут выделяться новые или важные термины, *курсивный шрифт* будет использоваться, чтобы обратить ваше внимание и для обозначения адресов URL, а моноширинный шрифт – для выделения программного кода на языке Python, например имен переменных и функций. Многострочные блоки программного кода или примеры команд будут оформлены моноширинным шрифтом в виде блоков:

```
>>> print "This is Python!"
This is Python!
```

В процессе создания этой книги и примеров приложений мы использовали три основные платформы – Mac OS X, Linux и Windows. Кроме того, мы использовали все основные броузеры (хотя не все они были представлены на снимках экрана), а именно: Firefox, Safari, Opera и Internet Explorer.

Ресурсы для книги

Обратиться к коллективу авторов можно по электронной почте *authors@withdjango.com*. На нашем веб-сайте, *http://withdjango.com*, содержится значительный объем вспомогательных материалов, ссылки на которые часто встречаются в книге.

Благодарности

Мое имя первым стоит в списке авторов, но эта книга не увидела бы свет без усилий других авторов. Пол и Уэсли – джентльмены и специалисты высочайшего класса, и совместная работа с ними стала для меня удивительным опытом.

Джентльменами и специалистами можно назвать всех разработчиков, составляющих ядро команды Django. Первоначальные авторы Django – Адриан Холовати (Adrian Holovaty), Якоб Каплан-Мосс (Jacob Kaplan-Moss), Саймон Уиллисон (Simon Willison) и Уилсон Майнер (Wilson Miner) – заложили (и продолжают развивать) удивительный фундамент, который был дополнен благодаря усилиям Малкольма Трединника (Malcolm Tredinnick), Джорджа Бауэра (Georg Bauer), Люка Планта (Luke Plant), Рассела Кейт-Маги (Russell Keith-Magee) и Роберта Уитамса (Robert Wittams). Каждый из этих парней поражает меня, а поразить меня совсем непросто.

Мне также хотелось бы поблагодарить двух сотрудников «Djangonauts» и ветеранов IRC – Кевина Менарда (Kevin Menard) и Джеймса Беннета (James Bennett), а также членов группы NYCDjango, где собрались удивительные и талантливые члены сообщества Django.

Наконец, огромное спасибо сотрудникам издательства Pearson, включая наших редакторов и технических рецензентов (Уэсли упомянет этих людей чуть ниже!), и особое спасибо техническим редакторам, чье внимание к деталям трудно переоценить.

Джефф Форсье (Jeff Forcier)
Нью-Йорк (штат Нью-Йорк)
Август 2008

Спасибо сообществам людей, сплотившихся вокруг Django, Python и других платформ, предназначенных для разработки веб-приложений и распространяемых с открытыми исходными текстами. Благодаря усилиям тысяч разработчиков и пользователей создаются мощные пакеты свободно распространяемого программного обеспечения.

Мои соавторы стали настоящей находкой и источником основных знаний и умений, а также проявили удивительную преданность делу. Несмотря на тот факт, что мы живем в разных концах континента, мне посчастливилось повстречать Джеффа и Уэса.

Хочу выразить благодарность группе разработчиков из Western Massachusetts Developers Group за интересные и жаркие дискуссии и огромный энтузиазм, проявленный к проекту книги.

Спасибо Джорджу Дж. Роса III (George J. Rosa III), президенту института фотографии Hallmark Institute of Photography, за то, что поддерживал меня и доверил мне выбор инструментов для работы, включая, конечно, и Django.

Летом 2008 года, после серьезной автомобильной аварии, я получил удивительный всплеск внимания и поддержки от моей семьи, друзей и сообщества. Для меня огромное значение имело все – и добрые пожелания, и оплата счетов, и деньги, и еда. В этих словах вы сами узнаете себя, и я еще раз благодарю вас.

И моей замечательной супруге Кетлин – спасибо тебе за внимание и понимание, поддержку и любовь.

Пол Биссекс (Paul Bissex)

Нортхемптон (штат Массачусетс)

Сентябрь 2008

Работа над второй моей книгой стала бесценным опытом. Я хотел бы поприветствовать двух моих соавторов, работа с которыми доставила мне огромное удовольствие. Они способны любого, имеющего некоторый опыт работы с языком программирования Python, привести в мир Django. Я счастлив тем, что сумел внести свой вклад в эту замечательную книгу о Django, и надеюсь, что в будущем мне доведется еще работать с ними. Было чрезвычайно приятно работать над этой книгой, как если бы это был открытый проект, используя те же инструменты, которые применяют разработчики каждый день для разработки программного обеспечения, изменяющего мир.

Я благодарен Дебре Уильямс Коли (Debra Williams Cauley) за помощь в управлении всем процессом с самого начала работы над этим проектом. Нас преследовали многочисленные изменения в составе сотрудников, но она позволила нам быть сосредоточенными только на рукописи. Хотя не было никаких гарантий, что будет создана книга о Django, которая будет пользоваться спросом, но она верила в наше стремление написать «правильную» книгу, которая будет востребована всем сообществом. Спасибо всем нашим техническим рецензентам – Майклу Торстону (Michael Thurston) (редактор-консультант по аудитории), Джо Блейлоку (Joe Blaylock) и Антонио Кангиано (Antonio Cangiano), а также всем, кто присылал свои отзывы Рафу Катсу (Rough Cuts), что позволило улучшить эту книгу по сравнению с первоначальным вариантом. Я также хотел бы выразить благодарность Мэтту Брауну (Matt Brown), лидеру проекта «Django Helper for Google App Engine», за его помощь в рецензировании приложения E, а также Эрику Уолстаду (Eric Walstad) и Эрику Эвенсону (Eric Evenson) за их заключительное рецензирование всей книги и комментарии.

Наконец, без поддержки наших семей эта книга была бы просто невозможна.

Уэсли Чан (Wesley Chun)

Кремниевая долина (штат Калифорния)

Август 2008

Введение

Если вы веб-разработчик, то есть программист, занимающийся созданием веб-сайтов, платформа Django легко может изменить вашу жизнь, как она изменила нашу.

Любой, кто обладает даже незначительным опытом разработки динамических веб-сайтов, знает, насколько сложно изобретать одно и то же снова и снова. Необходимо создать схему базы данных. Необходимо реализовать запись и извлечение данных из базы. Необходимо предусмотреть анализ адресов URL. Необходимо фильтровать данные, вводимые человеком. Необходимо создать инструменты редактирования информационного наполнения. Необходимо постоянно помнить о безопасности и удобстве использования. И так далее.

Как появились веб-платформы

В некоторый момент вы начинаете понимать, насколько это расточительно – тратить время на повторную реализацию одних и тех же особенностей в каждом новом проекте, и решаете создать свои собственные библиотеки с чистого листа или, что более вероятно, извлечь эти библиотеки из последнего и наиболее удачного проекта. После того как вы приступаете к работе над новым проектом, первое, что вы делаете – устанавливаете библиотеки. Благодаря этому вы экономите свои силы и время.

Однако начинают проявляться некоторые недостатки. Клиенты выражают желание получить функциональные возможности, отсутствующие в вашей библиотеке, поэтому вы добавляете в нее необходимый программный код. У разных клиентов возникают разные желания, в результате вы получаете несколько различных версий своей библиотеки, установленные на разных серверах. Сопровождение таких веб-приложений превращается в кошмар.

Поэтому, закаленный опытом, вы берете свою основную библиотеку и все лучшие дополнения из ваших проектов и объединяете их. Для большинства проектов вам уже не приходится изменять программный код своей библиотеки – вместо этого вы просто изменяете конфигура-

ционный файл. Ваша библиотека стала больше и сложнее, но при этом она обладает большими возможностями.

Примите наши поздравления – вы написали веб-платформу.

И пока вы (ваш коллектив, компания или ваши клиенты) продолжаете использовать библиотеку, вы несете ответственность за сохранение ее работоспособности. Не приведет ли к нарушениям в работе переход на новую версию операционной системы, веб-сервера или языка программирования? Обладает ли она достаточной гибкостью, чтобы в будущем в нее можно было вносить изменения без особых сложностей? Поддерживает ли она такие сложные, но нужные особенности, как управление сеансами, локализация и средства доступа к базам данных? А как насчет проверочных тестов?

Более простой путь

Вы взяли эту книгу в руки, потому что хотите отыскать более простой путь. Вам требуется мощная, гибкая, тщательно протестированная и первоклассная платформа для разработки веб-приложений, *но вам не хочется заниматься ее поддержкой.*

Вам необходим настоящий язык программирования – мощный, ясный, зрелый, хорошо документированный. Вам необходимо, чтобы для используемого языка программирования имелась обширная стандартная библиотека и широчайший выбор разнообразных и высококачественных пакетов сторонних разработчиков, начиная от пакетов создания файлов в формате CSV или круговых диаграмм и завершая пакетами реализации научных расчетов или обработки файлов изображений.

Вам необходима платформа, поддерживаемая энергичным, готовым прийти на помощь сообществом пользователей и разработчиков. Платформа, которая функционирует как хорошо отлаженная машина, но чьи компоненты слабо связаны между собой, благодаря чему вы легко можете менять их по мере необходимости.

Проще говоря, вам необходимы язык Python и платформа Django. Мы написали эту книгу, чтобы помочь вам изучить и начать пользоваться Django при решении реальных задач настолько легко, быстро и эффективно, насколько это возможно.

Мы уже не в Канзасе

Первоначально платформа Django была написана Адрианом Холоватй (Adrian Holovaty) и Саймоном Уиллисоном (Simon Willison), работавшим в World Online – семейной веб-компании, находившейся в городе Лоуренс, штат Канзас. Она появилась из-за необходимости быстро

разрабатывать приложения баз данных, наполняемых содержимым новостей.

После того как платформа Django доказала свою состоятельность, в июле 2005 года она была выпущена как проект с открытыми исходными текстами – это было время, когда, по иронии судьбы, было широко распространено мнение, что на языке Python реализовано *слишком мало* веб-платформ, – и быстро получила мощную поддержку. В настоящее время эта платформа является одним из лидеров не только среди платформ для разработки веб-приложений на языке Python, но и среди всех веб-платформ.

Конечно, платформа Django по-прежнему интенсивно используется компанией World Online, и некоторые из основных разработчиков платформы продолжают работать в этой компании и ежедневно пользуются платформой. Но, так как Django является программным продуктом, распространяемым с открытыми исходными текстами, большое число компаний и организаций по всему миру выбирают и используют ее в своих больших и маленьких проектах. В число этих компаний входят:

- The Washington Post
- The Lawrence Journal-World
- Google
- EveryBlock
- Newsvine
- Curse Gaming
- Tabblo
- Pownce

Безусловно, существуют тысячи других сайтов, созданных на основе Django, названия которых пока не так широко известны. Но по мере развития Django неизбежно будет увеличиваться число популярных сайтов, работающих на ее основе, и мы надеемся, что ваш сайт будет одним из них.

Разрабатывать веб-приложения лучше с использованием Python и Django

Разработка веб-приложений – это не самое простое дело. Вам приходится бороться с несовместимостью броузеров, со злонамеренными ботами, с ограничениями полосы пропускания и сервера и общей архитектурой, трудно поддающейся тестированию.

Конечно, мы полагаем, что наша книга является отличным введением в основы Django, но при этом мы стараемся не оставлять без внимания

упомянутые сложности – 20 процентов работы может потребовать 80 процентов времени. Мы работали со многими разработчиками и помогли многим из них в решении проблем, связанных с применением платформы Django, и мы не забыли их вопросы и держали их в уме при работе над этой книгой.

Если бы мы не считали Django и Python отличными продуктами, мы не стали бы брать на себя труд писать целую книгу о них. И когда мы будем подходить к каким-либо ограничениям или подвохам, о которых вам следует знать, мы сообщим об этом. Наша цель состоит в том, чтобы помочь вам довести работу до результата.

I

Введение

1. Практическое введение в Python для Django
2. Django для нетерпеливых: создание блога
3. Начало

1

Практическое введение в Python для Django

Добро пожаловать в Django, а также, кроме того, в Python! Прежде чем перейти к Django, мы дадим краткий обзор языка, который является основой приложений, разрабатываемых на платформе Django. Знакомство с другими языками программирования высокого уровня (C/C++, Java, Perl, Ruby и т. д.) упростит усвоение материала этой главы.

Однако, если вы никогда раньше не занимались программированием, сам язык Python прекрасно подходит на роль первого языка. В конце главы приводится список книг, которые можно использовать для обучения программированию на языке Python. Тем, кто плохо знаком с программированием, мы рекомендуем обратиться сначала к этим книгам, а затем вернуться сюда – это поможет вам извлечь больше пользы из следующих разделов.

В этой главе будет представлен язык программирования Python, причем особое внимание будет уделено основам языка и приемам программирования, которые имеют отношение к разработке приложений на платформе Django. Для эффективной работы с Django вам нужно знать не только основы языка Python, но также внутреннее устройство и принцип его действия, поэтому, когда будут рассматриваться некоторые особенности или требования платформы Django, мы не оставим без внимания то, что происходит за кулисами. Те, кто плохо знаком с языком Python или с программированием вообще, извлекут больше пользы, если получат общие сведения о Python из других источников, перед тем как приступать к чтению этой главы, или будут обращаться к ним в процессе ее чтения – какой стиль изучения подходит лучше, решать вам.

2 Зак. 1496

Практические приемы владения Python и Django

Django представляет собой высокоуровневую платформу, которая позволяет создавать веб-приложения, написав всего несколько строк программного кода. Эта платформа отличается простотой и гибкостью, позволяя без труда создавать собственные решения. Платформа Django написана на языке Python, объектно-ориентированном языке программирования, который соединяет в себе мощь таких языков системного программирования, как C/C++ и Java, с непринужденностью и скоростью разработки языков сценариев, таких как Ruby и Visual Basic. Это дает пользователям возможность создавать приложения, способные решать самые разнообразные задачи.

В этой главе будет показаны некоторые практические приемы программирования на языке Python, которыми должен владеть любой разработчик, использующий платформу Django. Вместо того чтобы пытаться воссоздать универсальный учебник по языку Python, мы сосредоточимся на тех концепциях языка, которые «должны быть» в арсенале программиста, использующего платформу Django. Фактически в этой главе будут приводиться фрагменты программного кода из самой платформы Django.

Python 2.x и Python 3.x

Во время нашей работы над этой книгой начался переход с версии Python 2.x на новое поколение версий Python, начиная с версии 3.0. Семейство 3.x не гарантирует обратную совместимость с предыдущими версиями языка, поэтому вполне возможно, что программный код, написанный для версии 2.x не будет работать с интерпретатором версии 3.x. Однако команда разработчиков Python стремится сделать переход на использование новой версии как можно более безболезненным; они предоставят надежные инструменты переноса сценариев с версии 2.x на версию 3.x, да и сам переход будет выполняться достаточно продолжительное время, поэтому никто не останется брошенным.

Команда разработчиков Python не планирует выполнить переход на версию 3.0 прямо сейчас – как и в большинстве основополагающих проектов, такой переход может оказаться разрушительным и должен выполняться с большой осторожностью, поэтому об этом переходе мы будем говорить лишь вскользь. Наиболее вероятно, что сама платформа Django будет переведена на новую версию, только когда основная масса пользователей (и вы в том числе!) будет готова к этому.

Введение: интерактивный интерпретатор языка Python

Интерактивный интерпретатор – это один из самых мощных инструментов, используемых в процессе разработки программ на языке Python, обеспечивая возможность тестирования фрагментов, состоящих всего из нескольких строк программного кода, без необходимости создавать, редактировать, сохранять и запускать файлы с исходными текстами. Более того, интерактивная оболочка интерпретатора Python проверяет корректность вводимого программного кода и позволяет опробовать различные вещи на новом программном коде – например, проверять структуры данных или изменять ключевые значения, прежде чем добавить его в файл с исходными текстами.

Во время чтения этой главы мы рекомендуем запустить интерактивную оболочку интерпретатора Python, чтобы тут же опробовать фрагменты программного кода – большинство интегрированных сред разработки на языке Python легко запускаются из командной строки или из меню приложений операционной системы. Используя интерактивную оболочку, вы получите непосредственный контакт с интерпретатором и быстрее овладеете Python и Django. Опытные программисты, использующие Python, такие как авторы этой книги, по-прежнему продолжают использовать интерактивную оболочку Python каждый день, даже обладая десятилетним опытом!

На протяжении всей книги вам будут встречаться фрагменты программного кода, начинающиеся со строки приглашения к вводу: >>>. Эти примеры можно опробовать непосредственно в интерактивной оболочке. Выглядят они примерно так, как показано ниже:

```
>>> print 'Hello World!'
Hello World!
>>> 'Hello World!'
'Hello World!'
```

Инструкция print – ваш лучший друг. Она не только может использоваться в приложениях для вывода информации, но и представляет собой бесценное средство отладки. Хотя часто бывает возможно вывести значение переменной без явного использования инструкции print, как это только что было продемонстрировано, но вы должны знать, что при этом нередко выводимые результаты отличаются от тех, что выводит инструкция print. В данном примере, используя инструкцию print, мы требуем от интерпретатора вывести *содержимое* строки, которое, конечно, не включает кавычки. Данное конкретное отличие касается только строк, для чисел таких различий не наблюдается.

```
>>> 10
10
>>> print 10
10
```

Однако для составных объектов, к которым мы подойдем ниже, различия могут оказаться более существенными – это обусловлено тем, что язык Python дает нам в руки полный контроль над тем, как должен вести себя объект при выводе с использованием и без использования инструкции print.

Подробнее о переменных и циклах будет говориться ниже и, тем не менее, чтобы получить некоторое представление о языке, взгляните на следующий, немного более сложный, фрагмент программного кода, в котором присутствует цикл for:

```
>>> for word in ['capitalize', 'these', 'words']:
...     print word.upper()
...
CAPITALIZE
THESE
WORDS
>>> for i in range(0, 5):
...     print i
...
0
1
2
3
4
```

Использование интерактивной оболочки при работе с Django

Очень удобно использовать интерактивную оболочку интерпретатора для экспериментов с программным кодом приложений, созданных на платформе Django, или с фрагментами реализации самой платформы. Но, если просто запустить интерпретатор и попытаться импортировать модули Django, будет получено сообщение об отсутствии переменной окружения DJANGO_SETTINGS_MODULE. Платформа Django предоставляет команду manage.py shell, которая выполняет все необходимые настройки окружения и позволяет избежать этой проблемы.

Команда manage.py shell по умолчанию использует оболочку iPython, если она установлена. Если же при установленной оболочке iPython вы все-таки хотите использовать стандартную интерактивную оболочку интерпретатора Python, используйте команду manage.py shell plain. В наших примерах мы будем использовать интерпретатор по умолчанию, но вам настоятельно рекомендуем пользоваться оболочкой iPython.

Важной особенностью языка Python является отказ от использования фигурных скобок ({}) для выделения блоков программного кода. Вместо скобок используются отступы: внутри данного фрагмента имеются различные уровни отступов. Обычно величина отступа составляет четыре пробела (хотя вы можете использовать любое число пробелов или символов табуляции). Если у вас есть опыт работы с другими языками программирования, может потребоваться некоторое время, чтобы привыкнуть к этой особенности, однако спустя короткое время вы поймете, что она не так плоха, как кажется на первый взгляд.

Последнее замечание об интерпретаторе: как только вы научитесь пользоваться интерактивной оболочкой, вы должны познакомиться с похожим инструментом, который называется iPython. Для тех, кто уже хорошо знаком с интерактивной оболочкой Python, можем сказать, что iPython – это еще более мощный инструмент! Он предоставляет множество таких возможностей, как доступ к командной оболочке операционной системы, нумерация строк, автоматическое оформление отступов, история команд и многое другое. Подробнее об iPython можно узнать на сайте *http://ipython.scipy.org*. Этот инструмент не распространяется в составе Python, но его свободно можно получить в Интернете.

Основы Python

В этом разделе мы познакомимся с некоторыми аспектами языка Python. Мы поговорим о комментариях, переменных, операторах и базовых типах данных. В следующих нескольких разделах мы еще ближе познакомимся с основными типами данных. Большая часть программного кода на языке Python (в том числе и программный код Django) находится в текстовых файлах с расширением `.py` – это стандартный способ сообщить системе, что это файл с программным кодом Python. Можно также встретить файлы с родственными расширениями, такими как `.pyc` или `.pyo` – они не будут вызывать проблем в системе и вы будете встречать их, но мы пока не будем отвлекаться на них.

Комментарии

Комментарии в языке Python начинаются с символа решетки (#). Если этот символ находится в начале строки с программным кодом, то вся строка является комментарием. Символ # может также появляться в середине строки, и тогда комментарием является часть от символа решетки и до конца строки. Например:

```
# эта строка целиком является комментарием
foo = 1          # короткий комментарий: переменной 'foo' присваивается число 1
print 'Python and %s are number %d' % ('Django', foo)
```

Комментарии используются не только для добавления пояснений к близлежащему программному коду, но и могут предотвращать вы-

полнение строк с программным кодом. Отличным примером такого способ использования комментариев могут служить файлы с настройками, такие как settings.py – параметры настройки, которые не нужны или имеют значения, отличные от значений по умолчанию, комментируются, благодаря чему их проще будет активировать вновь или сделать выбор конфигурационных параметров более очевидным.

Переменные и присваивание значений

Чтобы в языке Python использовать переменные, не требуется «объявлять» их тип, как в некоторых других языках программирования. Python – это язык программирования с «динамической типизацией». Переменные можно представить себе как имена, ссылающиеся на безымянные объекты, которые хранят фактические значения, то есть для любой переменной можно в любой момент изменить значение, как показано ниже:

```
>>> foo = 'bar'
>>> foo
'bar'
>>> foo = 1
>>> foo
1
```

В этом примере в переменную foo сначала записывается ссылка на строковый объект 'bar', а затем – на целочисленный объект 1. Примечательно, что строка, на которую ссылается переменная foo, исчезнет, если нет какой-то другой переменной, ссылающейся на нее (что вполне возможно!).

Так как имеется возможность присваивать именам другие значения, как в данном случае, никогда нельзя быть абсолютно уверенным в типе данных объекта, на который ссылается переменная в каждый конкретный момент времени, если не попытаться запросить эту информацию у интерпретатора. Однако, пока данная переменная ведет себя, как некоторый тип данных (например, если она обладает всеми методами строковых объектов), она может рассматриваться как экземпляр данного типа, даже если она имеет дополнительные атрибуты. Это называется грубым определением типов, или «утиной типизацией» – если это ходит как утка и крякает как утка, значит – это утка.

Операторы

Операторы являются довольно универсальной особенностью, и в языке Python присутствуют практически те же самые операторы, что и во многих других языках программирования. Сюда входят арифметические операторы, такие как +, -, * и т. д., а также соответствующие им *комбинированные операторы присваивания*, +=, -=, *= и т. д. Благодаря этому выражение x = x + 1 можно записать, как x += 1. В языке

Python отсутствуют операторы инкремента и декремента (++ и --), которые вы могли использовать в других языках.

Имеются также стандартные операторы сравнения, такие как <, >=, ==, != и т. д., которые можно объединять с помощью логических операторов and и or. Имеется также логический оператор not, инвертирующий логический результат сравнивания. Ниже показано, как можно объединять операторы сравнения с помощью логического оператора and:

```
show_output = True
if show_output and foo == 1:
    print 'Python and %s are number %d' % ('Django', foo)
```

Вы уже знаете, что для выделения блоков программного кода в языке Python используются отступы, а не фигурные скобки. Ранее говорилось, что благодаря отступам очень легко определить, какому блоку принадлежит та или иная строка программного кода. Но, кроме того, при такой организации синтаксиса проблема «повисшего else» становится *невозможной* просто потому, что принадлежность любого предложения else определенному оператору if становится достаточно очевидной.

Следует также отметить, что в Python вообще не используются некоторые символы. Не только фигурные скобки не нужны, но и символ точки с запятой (;), завершающий строку программного кода, и символ доллара ($), и круглые скобки (()), окружающие условные выражения (как это видно в предыдущем примере). Иногда вам может встретиться символ @, обозначающий декораторы, и многочисленные варианты использования символа подчеркивания (_). Создатель языка Python полагает, что чем меньше символов, загромождающих программный код, тем проще его читать.

Стандартные типы данных в языке Python

Теперь мы познакомим вас со стандартными типами данных, которые вам придется использовать при работе с платформой Django. Сюда входят скаляры и литералы (такие, как числа и строки), а также «контейнеры» и структуры данных, используемые для группировки нескольких объектов. Прежде чем перейти к основным типам данных, следует заметить, что все объекты в языке обладают одной особенностью – все они обладают некоторым логическим значением.

Логические значения объектов

Как и в большинстве других языков программирования, в Python существует всего два логических значения: True и False. Значение любого типа в языке Python может быть представлено в виде логического значения независимо от фактического значения. Например, любое значение числового типа, равное нулю, рассматривается как значение False в логическом контексте, а все ненулевые значения – как значе-

ние True. Точно так же пустые контейнеры интерпретируются как значение False, а непустые – как значение True.

Для определения логического значения любого объекта можно использовать функцию bool(). Кроме того, значения True и False сами по себе являются обычными значениями, которые можно явно присваивать переменным.

```
>>> download_complete = False
>>> bool(download_complete)
False
>>> bool(-1.23)
True
>>> bool(0.0)
False
>>> bool("")
False
>>> bool([None, 0])
True
```

Предыдущие примеры и результаты вызова функции bool() должны быть понятны. Но последний пример таит в себе одну хитрость: несмотря на то, что оба элемента списка в логическом контексте имеют значение False, тем не менее, непустой список рассматривается как значение True. Возможность получения значения «истинности» объектов используется, когда эти объекты играют роль *условного выражения* в таких инструкциях, как if или while, где ход выполнения программы зависит от логического значения этих объектов.

Обратите также внимание на значение None в последнем примере. Это специальное значение, эквивалентное значению NULL или void в других языках. Значение None в логическом контексте всегда интерпретируется как False.

Логические значения являются литералами, так же как и числа, которые рассматриваются в следующем разделе.

Числа

В языке Python имеется два элементарных числовых типа: int (целые числа) и float (числа с плавающей точкой). В соответствии с мантрой «Простое лучше сложного», в языке Python имеется всего один целочисленный тип данных, int, в отличие от многих других языков программирования, имеющих несколько целочисленных типов.[1] В допол-

[1] Первоначально в языке Python имелся еще один целочисленный тип, который назывался long, но в настоящее время его функциональность была объединена в тип int. Однако в устаревшем программном коде и документации до сих пор можно увидеть завершающий символ «L», использовавшийся для представления длинных целых чисел, например: 1L, -42L, 99999999999999999L и т. д.

нение к обычной, десятичной форме записи, целые числа могут записываться в шестнадцатеричной (по основанию 16) и восьмеричной (по основанию 8) системах счисления. Тип `float` представляет вещественные числа с плавающей точкой двойной точности и должен быть знаком вам по другим языкам программирования. Ниже приводится несколько примеров целых чисел и чисел с плавающей точкой, а также некоторые операторы, применяемые к ним:

```
>>> 1.25 + 2.5
3.75
>>> -9 - 4
-13
>>> 1.1
1.1000000000000001
```

Ой! А что это получилось в последнем примере? Тип данных `float` может использоваться для представления огромного диапазона чисел, однако он не обладает высокой точностью, в терминах представления рациональных чисел с дробной частью в периоде. По этой причине был создан еще один тип данных, с именем `Decimal`, – для представления чисел с плавающей точкой; он не является встроенным типом данных и доступен в виде модуля `decimal`. Этот тип данных способен представлять более ограниченный диапазон данных, но с более высокой точностью. В языке Python имеется также встроенный числовой тип данных `complex`, который может использоваться в научных расчетах.

В табл. 1.1 перечисляются числовые типы данных, а также приводится несколько примеров.

Таблица 1.1. Встроенные числовые типы данных

Тип	Описание	Примеры
`int`	Целые числа со знаком (неограниченного размера)	`-1, 0, 0xE8C6, 0377, 42`
`float`	Числа с плавающей точкой двойной точности	`1.25, 4.3e+2, -5., -9.3e, 0.375`
`complex`	Комплексные числа (вещественная часть + мнимая)	`2+2j, .3-j, -10.3e+5-60j`

Числовые операторы

Числовые типы данных поддерживают основные арифметические операции, с которыми вы должны быть знакомы по другим языкам программирования: сложение (+), вычитание (-), умножение (*), деление (/ и //), деление по модулю (%) и возведение в степень (**).

Оператор деления / представляет операцию «классического деления» в том смысле, что он усекает дробную часть результата, когда в операции участвуют два целых числа, и операцию «истинного деления»,

когда в операции участвуют числа с плавающей точкой. В языке Python имеется также явный оператор «деления с усечением дробной части», который всегда возвращает целое число независимо от типов операндов:

```
>>> 1 / 2          # деление с усечением дробной части (операнды типа int)
0
>>> 1.0 / 2.0      # true division (операнды типа float)
0.5
>>> 1 // 2         # деление с усечением дробной части (оператор //)
0
>>> 1.0 // 2.0     # деление с усечением дробной части (оператор //)
0.0
```

Наконец, к целым числам могут применяться битовые операторы И (&), ИЛИ (|), ИСКЛЮЧАЮЩЕЕ-ИЛИ (^) и инверсия (~), а также операторы сдвига влево и вправо (<< и >>) и соответствующие им комбинированные операторы присваивания, такие как &=, <<= и т. д.

Встроенные и фабричные функции для работы с числами

Для каждого числового типа имеется *фабричная* функция, которая позволяет выполнять преобразование из одного числового типа в другой. Некоторые читатели могут сказать: не «преобразование», а «приведение», но мы не используем этот термин в языке Python, поскольку здесь не выполняется *изменение* типа существующего объекта. Функция возвращает новый объект, созданный на основе первоначального (откуда и взялось название «фабричная»). Вызвав функцию `int(12.34)`, легко можно создать целочисленный объект со значением 12 (с ожидаемым усечением дробной части), вызов `float(12)` вернет значение `12.0`. Наконец, у нас имеются типы `complex` и `bool`.

В языке Python имеется также несколько *встроенных* функций, применяемых к числам, такие как `round`, выполняющая округление чисел с плавающей точкой до указанного числа знаков после запятой, или `abs`, возвращающая абсолютное значение числа. Ниже приводятся примеры использования этих и других встроенных функций, предназначенных для работы с числами:

```
>>> int('123')
123
>>> int(45.67)
45
>>> round(1.15, 1)
1.2
>>> float(10)
10.0
>>> divmod(15, 6)
(2, 3)
```

```
>>> ord('a')
97
>>> chr(65)
'A'
```

За дополнительной информацией по этим и другим числовым функциям обращайтесь к главе «Numbers» в книге «Core Python Programming» (Prentice Hall, 2006) или к другой справочной литературе, можно также обратиться к документации по языку Python в Интернете. Теперь рассмотрим строки и другие основные контейнерные типы данных в языке Python.

Последовательности и итерируемые объекты

Во многих языках программирования имеются такие структуры данных, как массивы, которые обычно имеют фиксированный размер и содержат группу объектов одного типа, доступных по индексу. Последовательности в языке Python играют ту же роль, но могут содержать объекты разных типов, а также увеличиваться или уменьшаться в размерах. В этом разделе мы рассмотрим два наиболее популярных типа данных в языке Python: **списки** (`[1, 2, 3]`) и **строки** (`'python'`). Они являются частью обширного множества структур данных, называемых **последовательностями**.

Последовательности являются одним из представителей **итерируемых объектов** – структур данных, по элементам которых можно выполнять «обход» или «итерации». Главная особенность итерируемых объектов состоит в том, что вы имеете возможность обращаться к ним, запрашивая доступ к следующему элементу с помощью метода `next`, который продолжает читать внутреннюю коллекцию своих объектов, пока не исчерпает ее. Последовательности в языке Python поддерживают не только последовательный способ доступа (хотя в 99 процентах случаев вы будете использовать циклы `for` вместо метода `next`), но и произвольный, что дает возможность получать доступ к определенному объекту в последовательности. Например, выражение `my_list[2]` вернет третий элемент списка (индексирование элементов последовательностей начинается с 0).

Третий тип последовательностей называется **кортежем**. Кортежи легко можно описать фразой: «списки с ограниченными возможностями, доступные только для чтения», вследствие чего они используются совершенно для других целей. Они вряд ли станут основной структурой данных в ваших приложениях, но мы должны рассказать, что они из себя представляют и для чего используются. Поскольку вы уже наверняка знаете, что такое строки, мы сначала рассмотрим списки, а кортежи обсудим в последнюю очередь. В табл. 1.2 перечислены все обсуждаемые типы последовательностей и даются некоторые примеры.

Таблица 1.2. Примеры использования последовательностей

Тип	Примеры
str	'django', '\n', "", "%s is number %d" % ('Python', 1), """hey there"""
list	[123, 'foo', 3.14159], [], [x.upper() for x in words]
tuple	(456, 2.71828), (), ('need a comma even with just 1 item',)

Извлечение срезов последовательностей

Чуть выше упоминалось, что существует возможность прямого обращения к элементам последовательностей по их индексам. Ниже приводятся несколько примеров такого способа обращения к строкам. В отличие от многих других языков, строки в языке Python могут рассматриваться и как самостоятельные объекты, и как списки отдельных символов.

```
>>> s = 'Python'
>>> s[0]
'P'
>>> s[4]
'o'
>>> s[-1]
'n'
```

В языке Python допускается также использовать отрицательные индексы. Наверняка вам приходилось использовать выражение data[len(data)-1] или data[data.length-1], чтобы получить доступ к последнему элементу какого-нибудь массива. Как показывает последний пример в предыдущем фрагменте, для этого достаточно использовать индекс -1.

Точно так же с помощью индексов можно обратиться сразу к нескольким элементам последовательности – в языке Python эта операция называется **извлечением среза**. В операции извлечения среза участвует пара индексов, разделенных двоеточием (:), – скажем, *i* и *j*. Когда производится попытка извлечь срез последовательности, интерпретатор возвращает подмножество элементов, начиная с первого индекса *i* и заканчивая вторым индексом *j*, но не включая элемент с этим индексом в срез.

```
>>> s = 'Python'
>>> s[1:4]
'yth'
>>> s[2:4]
'th'
>>> s[:4]
'Pyth'
>>> s[3:]
'hon'
>>> s[3:-1]
'ho'
>>> s[:]
```

```
'Python'
>>> str(s)
'Python'
```

Отсутствие индекса означает «от начала» или «до конца» – в зависимости от того, какой индекс отсутствует. Ложный срез (возвращающий копию[1] всей последовательности) можно получить, используя оператор индекса `[:]`. И напоследок: запомните, что, хотя во всех предыдущих примерах операции выполнялись над строками, тем не менее, синтаксис извлечения срезов в равной степени применим к спискам и другим типам последовательностей.

Прочие операторы последовательностей

В предыдущем разделе мы видели выполнение операций извлечения среза с использованием операторов `[]` и `[:]`. Но над последовательностями можно выполнять и другие операции, включая конкатенацию (+), дублирование (*), а также проверку на вхождение (`in`) или не вхождение (`not in`). Как и прежде, в наших примерах будут использоваться строки, но эти операции могут выполняться и над другими последовательностями.

Альтернативные способы выполнения конкатенации

Мы рекомендуем избегать использования оператора + для объединения последовательностей. Пока вы плохо знакомы с языком Python, этот оператор поможет решить проблему объединения пары строк, однако это решение снижает производительность. (Объяснение причин этого привело бы к необходимости объяснять реализацию интерпретатора Python на языке C, что выходит за рамки этой книги, поэтому просто поверьте нам на слово.)

Например, в случае строк вместо выражения `'foo'+'bar'` можно использовать оператор форматирования строк (%), который будет рассматриваться в следующем разделе, – `'%s%s' % ('foo', 'bar')`. Другой способ объединения строк, особенно когда они находятся в списке, состоит в использовании строкового метода `join`, например: `''.join(['foo', 'bar'])`. У списков имеется метод `extend`, добавляющий содержимое другого списка в текущий (аналог операции `list1 += list2`), при этом операция `list1.extend(list2)` выполняется быстрее.

[1] Когда мы говорим «копия», мы подразумеваем копию ссылок, а не самих объектов. Более корректно такие копии называть поверхностными копиями. Дополнительные сведения о копировании изменяемых объектов приводятся в следующем разделе.

```
>>> 'Python and' + 'Django are cool!'
'Python andDjango are cool!'
>>> 'Python and' + ' ' + 'Django are cool!'
'Python and Django are cool!'
>>> '-' * 40
'----------------------------------------'
>>> 'an' in 'Django'
True
>>> 'xyz' not in 'Django'
True
```

Списки

В языке Python имеется тип данных, который во многом действует как массивы в других языках программирования, – это список. Списки являются последовательностями, обеспечивающими возможность изменения своего содержимого и размера, и могут хранить объекты любых типов. Ниже приводится пример, демонстрирующий, как можно создать список и какие операции можно выполнять над ним:

```
>>> book = ['Python', 'Development', 8]     # 1) создать список
>>> book.append(2008)                       # 2) добавить объект в конец
>>> book.insert(1, 'Web')                   # 3) вставить объект
>>> book
['Python', 'Web', 'Development', 8, 2008]
>>> book[:3]                          # 4) получить первые три элемента
['Python', 'Web', 'Development']
>>> 'Django' in book                  # 5) объект присутствует в списке?
False
>>> book.remove(8)                    # 6) удалить объект
>>> book.pop(-1)                      # 7) удалить объект по индексу
2008
>>> book
['Python', 'Web', 'Development']
>>> book * 2                                # 8) дублировать
['Python', 'Web', 'Development', 'Python', 'Web', 'Development']
>>> book.extend(['with', 'Django'])         # 9) объединить списки
>>> book
['Python', 'Web', 'Development', 'with', 'Django']
```

Ниже кратко описывается, что происходит в этом примере:

1. Создается список, содержащий пару строк и целое число.
2. В конец списка добавляется еще одно целое число.
3. В позицию с индексом 1 вставляется строка.
4. Извлекается срез, содержащий первые три элемента.
5. Проверяется проверка вхождения объекта в список. (Присутствует ли элемент в списке?)
6. Из списка удаляется конкретный объект независимо от его местоположения в списке.

7. Удаляется (и возвращается) элемент с заданным индексом.
8. Демонстрируется применение оператора дублирования *.
9. Список дополняется содержимым другого списка.

Как видите, списки представляют собой весьма гибкие объекты. А теперь рассмотрим методы списков.

Методы списков

Давайте вернем список в состояние, в каком он был в середине предыдущего набора примеров. А теперь отсортируем список и немного поговорим на эту тему.

```
>>> book = ['Python', 'Web', 'Development', 8, 2008]
>>> book.sort()     # ВНИМАНИЕ: сортировка выполняется непосредственно
                    # в списке возвращаемое значение отсутствует!
>>> book
[8, 2008, 'Development', 'Python', 'Web']
```

«Сортировка» объектов разных типов – это действительно нечто необычное. Как можно сравнивать объекты (например, строки с числами), которые невозможно сравнивать? В языке Python используется алгоритм «наилучшего предположения» относительно того, что «считается правильным»: сначала сортируются все числовые значения (в порядке возрастания), а затем строки – в алфавитном порядке. Возможно, этот пример покажется не очень осмысленным, но стоит начать оперировать файлами и экземплярами классов, как определенный смысл начинает появляться.

Встроенные методы списков, такие как `sort`, `append` и `insert`, изменяют объект непосредственно и не имеют возвращаемого значения. Начинающим программистам кажется странным, что метод `sort` не возвращает отсортированную копию списка, и потому они остерегаются использовать его. Напротив, строковый метод `upper`, который мы видели выше, возвращает строку (копию оригинальной строки, в которой все символы приведены К ВЕРХНЕМУ РЕГИСТРУ). В отличие от списков, строки *не* являются изменяемыми объектами, и по этой причине метод `upper` возвращает (модифицированную) копию. Подробнее об изменяемости объектов рассказывается ниже.

Конечно, часто бывает желательно получить отсортированную копию заданного списка, а не сортировать сам список. В версии Python 2.4 и выше имеются встроенные функции `sorted` и `reversed`, которые принимают список в виде аргумента и возвращают отсортированную (по возрастанию или по убыванию) копию.

Генераторы списков

Генератор списков – это конструкция (заимствованная из языка программирования Haskell), содержащая программный код, который создает список, содержащий значения/объекты, сгенерированные про-

граммным кодом. Например, предположим, что у нас имеется список, содержащий целые числа от 0 до 9, и нам необходимо увеличить каждое число на 1 и вернуть результат в виде списка. Используя генератор списков, это легко можно реализовать, как показано ниже.

```
>>> data = [x + 1 for x in range(10)]
>>> data
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Генераторы списков подобно обычным спискам заключаются в квадратные скобки и представляют собой сокращенную версию цикла `for`. Хотя циклы мы еще не рассматривали – мы подойдем к ним вскоре – вы можете видеть, насколько легко читается генератор списков. Первая часть – это выражение, генерирующее элементы списка результата, а вторая часть – цикл по входному выражению (которое должно возвращать последовательность).

«Читать генераторы списков» рекомендуется так: начать с внутреннего цикла `for` и затем скользить взглядом вправо, к условной инструкции `if`, если таковая присутствует (в нашем первом примере нет ни одной условной инструкции), и затем отображать выражение, расположенное в начале генератора списков, на каждый полученный элемент. Проверьте, сумеете ли вы прочитать следующий генератор списков.

```
>>> even_numbers = [x for x in range(10) if x % 2 == 0]
>>> even_numbers
[0, 2, 4, 6, 8]
```

Этот второй пример демонстрирует использование дополнительной, фильтрующей условной инструкции `if`, расположенной в самом конце. Кроме того, в нем отсутствует какая-либо логика, изменяющая значения – выражение `x` само по себе является допустимым выражением (которое, конечно же, возвращает само значение `x`). Такие генераторы списков удобно использовать для фильтрации последовательностей.

Выражения-генераторы

В языке Python имеется еще одна конструкция, напоминающая генераторы списков, которая называется **выражение-генератор**. Она функционирует примерно так же, как и генератор списков, за исключением того, что выполняет так называемые отложенные вычисления. Вместо того чтобы обрабатывать и создавать структуру данных целиком, со всеми объектами, заключенными в нее, выражение-генератор создает по одному объекту за раз, что способствует уменьшению объема потребляемой памяти (и при этом иногда дает выигрыш в скорости).

В нашем последнем примере мы использовали генератор списков для выбора четных чисел из списка, содержащего десять чисел, но что если список содержит не десять, а десять тысяч или десять миллионов чисел? Что если список содержит не просто целые числа, а сложные или большие структуры данных? В таких случаях экономное отноше-

ние к памяти, свойственное выражениям-генераторам, может сэкономить целый день. Мы можем немного изменить генератор списков, чтобы превратить его в выражение-генератор, для чего достаточно заменить квадратные скобки круглыми.

```
>>> even_numbers = (x for x in range(10000) if x % 2 == 0)
>>> even_numbers
<generator object at 0x ...>
```

Выражения-генераторы впервые появились в версии Python 2.4, поэтому, если вы работаете с версией Python 2.3, вы не сможете пользоваться ими, а кроме того, выражения-генераторы до сих пор не заняли достойного положения в коллективном сознании программистов на языке Python. Однако в любой ситуации, когда есть вероятность, что исходная последовательность будет иметь гигантский размер, гораздо практичнее будет использовать выражения-генераторы вместо генераторов списков.

Строки

Другим типом последовательностей в языке Python является строка, которую можно интерпретировать как массив символов, заключенный в апострофы или в кавычки (`'this is a string'` или `"this is a string"`). Кроме того, в отличие от списков, строки не могут изменять содержимое или размер. В ходе операции, изменяющей строку, в действительности создается новая строка, отличная от оригинала. Однако при обычном использовании это обстоятельство никак не ощущается и обретает особую значимость, только когда приходится сталкиваться с проблемами нехватки памяти или с похожими проблемами.

Подобно спискам строки также имеют методы, но, опять же из-за того, что строки являются неизменяемыми объектами, ни один из имеющихся методов не модифицирует саму строку, а возвращает модифицированную копию. В то время, когда мы работали над этой книгой, строки имели не менее *37* методов! Мы рассмотрим только те из них, которые наиболее вероятно будут использоваться в приложениях на платформе Django. Ниже приводятся несколько примеров.

```
>>> s = 'Django is cool'          # 1
>>> words = s.split()             # 2
>>> words
['Django', 'is', 'cool']
>>> ' '.join(words)               # 3
'Django is cool'
>>> '::'.join(words)              # 4
'Django::is::cool'
>>> ''.join(words)                # 5
'Djangoiscool'
>>> s.upper()                     # 6
'DJANGO IS COOL'
>>> s.upper().isupper()           # 7
True
```

```
>>> s.title()                          # 8
'Django Is Cool'
>>> s.capitalize()                     # 9
'Django is cool'
>>> s.count('o')                       # 10
3
>>> s.find('go')                       # 11
4
>>> s.find('xxx')                      # 12
-1
>>> s.startswith('Python')             # 13
False
>>> s.replace('Django', 'Python')      # 14
'Python is cool'
```

Ниже кратко описывается то, что происходит в предыдущих примерах:

1. Создается первоначальная строка.
2. Выполняется разбиение строки по любым пробельным символам и превращение ее в список подстрок.
3. Выполняется действие, обратное действию, представленному в примере #2. (Подстроки из списка объединяются в единую строку и разделяются пробелами.)
4. То же, что и в примере #3, но подстроки отделяются друг от друга парой двоеточий.
5. То же, что и в примере #3, но подстроки не отделяются друг от друга (все подстроки сливаются в одну строку).
6. Создается (и тут же уничтожается) новая строка, представляющая собой копию оригинальной строки, в которой все символы приведены к верхнему регистру [смотрите также метод `lower`].
7. Демонстрирует объединение методов в цепочку и подтверждает, что во вновь созданной строке все символы приведены к верхнему регистру [смотрите также метод `islower` и другие].
8. Преобразует строку так, что каждое отдельное слово в ней начинается с символа верхнего регистра, а все остальные символы преобразуются в символы нижнего регистра.
9. Приводит первый символ строки к верхнему регистру, а остальные – к нижнему.
10. Подсчитывает, сколько раз встречается в строке подстрока `'o'`.
11. Определяет индекс подстроки `'go'` в строке (4) [смотрите также метод `index`].
12. То же, что и в примере #11. Демонстрирует, что в случае отсутствия совпадений возвращается значение `-1`.
13. Проверяет, начинается ли строка с указанной подстроки; в данном случае – нет [смотрите также метод `endswith`].
14. Простой поиск с заменой.

Существует еще один метод – splitlines, напоминающий метод split, который отыскивает символы перевода строки (вместо пробельных символов). Если у вас имеется строка, содержащая эти символы, например полученная в результате чтения содержимого текстового файла, можно воспользоваться методом rstrip, чтобы удалить завершающие пробельные символы (или даже метод strip, чтобы удалить начальные *и* завершающие пробельные символы).

Пример #7 демонстрирует, как можно объединять методы в цепочки, при условии что вы точно знаете, какой тип объекта каждый из них возвращает. Поскольку заранее известно, что метод upper возвращает строку, допускается тут же вызывать другой строковый метод этой новой строки. В данном случае вызывается метод isupper, который определяет, состоит ли (или нет) строка только из символов верхнего регистра. Например, если бы возвращаемый объект был списком, а не строкой, то можно было бы вызвать метод списка.

Помимо метода isupper существует еще множество методов, имена которых начинаются с префикса is, такие как isalnum, isalpha и т. д. В табл. 1.3 приводится краткое описание методов, представленных в этом разделе. Как вы уже наверняка догадались, существует множество других методов, поэтому за полной информацией о строковых методах мы отсылаем вас к вашей любимой книге по языку Python.

Таблица 1.3. Популярные строковые методы в языке Python

Строковый метод	Описание
count	Количество вхождений подстроки в строку
find	Выполняет поиск подстроки [смотрите также методы index, rfind, rindex]
join	Объединяет подстроки в одну строку
replace	Выполняет поиск и замену (под)строки
split	Разбивает строку на подстроки по указанному символу-разделителю [смотрите также метод splitlines]
startswith	Определяет, начинается ли строка с указанной подстроки [смотрите также метод endswith]
strip	Удаляет ведущие и завершающие пробельные символы [смотрите также методы rstrip, lstrip]
title	Приводит к верхнему регистру первый символ каждого слова в строке [смотрите также методы capitalize, swapcase]
upper	Приводит к верхнему регистру все символы в строке [смотрите также метод lower]
isupper	Проверяет, состоит ли строка только из символов верхнего регистра [смотрите также методы islower и т. д.]

Спецификаторы строк

Строки в языке Python позволяют указывать дополнительные спецификаторы перед открывающей кавычкой: `r` – для «сырых» строк и `u` – для строк Юникода. Эти спецификаторы используются как в программном коде, так и при выводе результатов в интерактивной оболочке интерпретатора.

```
>>> mystring = u'This is Unicode!'
>>> mystring
u'This is Unicode!'
```

При этом при выводе «сырых» строк или строк Юникода инструкцией `print` спецификатор не выводится:

```
>>> mystring = u'This is Unicode!'
>>> print mystring
This is Unicode!
>>> str(mystring)
'This is Unicode!'
```

Спецификатор `r` сообщает интерпретатору, что он не должен выполнять преобразование каких-либо специальных символов в строке. Так, специальный символ `\n` обычно обозначает перевод строки, но иногда бывает необходимо, чтобы такое преобразование *не выполнялось*, – например в именах файлов DOS: `filename = r'C:\temp\newfolder\robots.txt'`.

Другая область использования «сырых» строк – регулярные выражения, из-за интенсивного использования в них специальных символов, таких как обратный слеш (\). В разделе, описывающем регулярные выражения, мы будем записывать регулярные выражения в виде «сы-

«Сырые» строки в Django

«Сырые» строки часто можно встретить в программном коде на языке Python, где используются регулярные выражения. При работе с Django «сырые» строки используются в настройках правил обработки адресов URL, посредством которых платформа Django определяет, какой части приложения передать управление, на основе соответствия запрашиваемого адреса URL тому или иному регулярному выражению, определяемому вами. Использование «сырых» строк для записи этих правил обеспечивает более удобочитаемый их вид, а чтобы соблюсти непротиворечивость, «сырые» строки обычно используются для записи любых регулярных выражений независимо от наличия в них символа обратного слеша.

рых» строк r'\w+@\w+\.\w+', так как в этом случае они проще читаются, чем при использовании обычных (и потому экранированных) строк: '\\w+@\\w+\\.\\w+'.

Поскольку обычные строки в языке Python, как правило, могут содержать только ограниченный набор символов – западный алфавит плюс несколько специальных символов, – они не могут содержать все диапазоны символов, которые используются в языках, отличных от английского. Юникод – это новейшая кодировка, обеспечивающая поддержку чрезвычайно широкого диапазона символов, в которой подобные ограничения отсутствуют. Разработчиками платформы Django совсем недавно (на момент написания этих строк) предпринимались определенные шаги, чтобы обеспечить поддержку Юникода во всех компонентах платформы. Используя платформу Django для разработки приложений, вы увидите массу объектов строк Юникода.

Оператор форматирования строк и тройные кавычки

В предыдущих примерах в этой главе вы уже видели оператор форматирования строк (%) – он используется для подготовки исходных данных различных типов к выводу в виде строки с помощью *строки формата*, которая содержит специальные директивы. Ниже приводится еще один пример, демонстрирующий это.

```
>>> '%s is number %d' % (s[:6], 1)
'Django is number 1'
>>> hi = '''hi
there'''
>>> hi
'hi\nthere'
>>> print hi
hi
there
```

В предыдущем примере у нас имеется строка (которой соответствует директива форматирования %s) и целое число (%d). Оператор форматирования объединяет строку формата слева с кортежем (не списком) аргументов справа – аргументы и директивы должны в точности соответствовать друг другу. За подробной информацией обо всех директивах форматирования мы отсылаем вас к другим доступным вам источникам по языку Python.

Тройные кавычки, применение которых также демонстрируется в предыдущем примере, являются уникальной особенностью языка Python и позволяют встраивать в строки специальные символы, не используя специальные обозначения. Если у вас имеется длинная строка, которую необходимо воспроизвести, вам больше не придется беспокоиться о расстановке символов конца строки, таких как \\n или \\r\\n; вместо этого можно воспользоваться тройными кавычками, как показано в следующем фрагменте XML.

```
xml = '''
<?xml version="1.0"?>
<Request version="%.1f">
    <Header>
        <APIName>PWDDapp</APIName>
        <APIPassword>youllneverguess</APIPassword>
    </Header>
    <Data>
        <Payload>%s</Payload>
        <Timestamp>%s</Timestamp>
    </Data>
</Request>
'''
```

Наконец, обратите внимание, что в предыдущем примере используют ся директивы форматирования, но отсутствует оператор форматирова ния и кортеж с аргументами. Оператор форматирования строк – это самый обычный оператор, поэтому имеется возможность определить строку формата в одной части программы, а использовать ее, вместе с оператором форматирования и кортежем аргументов, – в другой.

```
import time          # обеспечить доступ к функции time.ctime()
VERSION = 1.2        # установить номер версии приложения

# [...]

def sendXML(data):  # определение некоторой функции sendXML()
    'sendXML() - transmit XML data'

# [...]

payload = 'super top-secret information'
sendXML(xml % (VERSION, payload, time.ctime()))
```

Кортежи

Кортежи являются близкими родственниками списков, которые обсуждались несколькими разделами выше. Одно из очевидных отличий состоит в том, что списки заключаются в квадратные скобки, а кортежи – в круглые, но, помимо этого, нам потребуется вернуться к рассмотрению модели объектов в языке Python. В отличие от списков, которые позволяют изменять их значения и обладают методами, выполняющими эти изменения, кортежи относятся к категории неизменяемых объектов, то есть не позволяют изменять свои значения и, отчасти по этой причине, не имеют методов.

Столкнувшись с кортежами впервые, начинающие программисты обычно задают вопрос: зачем нужен этот отдельный тип данных или, другими словами, почему бы просто не использовать списки «только для чтения»? На первый взгляд, они правы, однако на кортежи возложена не только роль списков, доступных «только для чтения». Главное назначение кортежей – передавать параметры функциям (при их

вызове) и предотвратить возможность изменения их содержимого посторонними функциями.

Это совершенно не означает, что в остальных случаях они бесполезны. Кортежи редко используются самими программистами, но они очень часто используются внутренними механизмами интерпретатора. Например, массу кортежей можно увидеть в конфигурационных файлах Django. Пусть у них нет методов, но, тем не менее, к ним могут применяться встроенные функции и обычные операторы, предназначенные для работы с последовательностями.

Проблемы в Django, связанные с кортежами

Кортежи часто можно встретить в типичных приложениях на платформе Django. Начинающим программистам кортежи кажутся немного странной разновидностью последовательностей, например, при определении кортежа, состоящего из одного элемента, *обязательно* должна присутствовать завершающая запятая. Взгляните на следующие примеры и попробуйте понять, что в них происходит:

```
>>> a = ("one", "two")
>>> a[0]
'one'
>>> b = ("just-one")
>>> b[0]
'j'
>>> c = ("just-one",)
>>> c[0]
'just-one'
>>> d = "just-one",
>>> d[0]
'just-one'
```

Что не так во втором примере? Запомните, не круглые скобки, а запятые задают кортеж. Поэтому переменная `b` представляет собой обычную строку и поэтому при обращении к элементу `b[0]` возвращается первый символ этой строки, как показано в примере. Завершающая запятая в инструкции присваивания переменной `c` образует кортеж, поэтому при обращении к элементу `c[0]` возвращается ожидаемое значение. Можно даже вообще отбросить круглые скобки, как это сделано в инструкции присваивания значения переменной `d`, но лучше этого не делать. Один из основных принципов языка Python гласит, что лучше писать явные определения, чем полагаться на неявное поведение.

Многие значения конфигурационных параметров Django определены в виде кортежей: параметры администрирования, правила работы с адресами URL и многие настройки в файле `settings.py`. Некоторые компоненты платформы Django могут даже сообщать о том, что сделано неправильно. Если в виде значения одного из параметров администратора определить строку, когда предполагается, что значением будет

кортеж, то можно получить полезное сообщение – например, такое: "admin.list_display", if given, must be set to a list or tuple (значением параметра "admin.list_display" должен быть список или кортеж). С другой стороны, если в значениях параметров настройки ADMINS или MANAGERS будет отсутствовать завершающая запятая, вы обнаружите, что сервер пытается отправить электронные письма по адресам, состоящим из отдельных символов первого имени! Поскольку с этой проблемой очень часто сталкиваются начинающие разработчики приложений на платформе Django, мы еще вернемся к ней в разделе «Типичные проблемы».

Встроенные и фабричные функции последовательностей

Подобно числам все типы последовательностей имеют специальные фабричные функции, которые создают экземпляры требуемых типов: list, tuple или str. Кроме того, строки Юникода могут также создаваться с помощью функции unicode. Обычно функция str отвечает за воспроизведение объекта в удобочитаемом для человека, или *печатном*, виде. В языке Python существует похожая функция с именем repr, но в отличие от функции str она воспроизводит *вычисленное* строковое представление объекта. Как правило, это такое строковое представление объекта, которое позволяет получить исходный объект с помощью инструкции eval.

Встроенная функция len возвращает количество элементов в последовательности. Функции max и min возвращают «наибольший» и «наименьший» элемент последовательности соответственно. Еще одна пара функций – any и all – проверяют, имеет ли хотя бы один или все элементы последовательности значение True.

Вы уже видели, как в языке Python функция range помогает организовать обход последовательности целых чисел в цикле for, который в других языках программирования изначально обладает такой возможностью, тогда как Python больше ориентирован на применение итераторов. Встроенная функция enumerate объединяет оба стиля – она возвращает специальный итератор, который обеспечивает последовательный доступ к индексам элементов и к элементам, соответствующим этим индексам.

Функции, общие для всех последовательностей, перечислены в табл. 1.4.

Таблица 1.4. Встроенные и фабричные функции последовательностей

Функция	Описание*
str	(Печатаемое) строковое представление [смотрите также функции repr, unicode]
list	Представляет объект в виде списка
tuple	Представляет объект в виде кортежа

Функция	Описание[a]
`len`	Возвращает количество элементов в объекте
`max`	Возвращает «наибольший» элемент последовательности [смотрите также функцию `min`]
`range`	Возвращает последовательность чисел в указанном диапазоне [смотрите также функции `enumerate`, `xrange`]
`sorted`	Сортирует список элементов [смотрите также функцию `reversed`]
`sum`	Суммирует (числовые) значения элементов последовательности
`any`	Проверяет, имеется ли в последовательности хотя бы один элемент, который в логическом контексте имеет значение `True` [смотрите также функцию `all`]
`zip`	Итератор кортежей из N элементов, в каждом из которых объединяются соответствующие элементы из N последовательностей

[a] Несмотря на то, что в описаниях функций говорится, что они применяются к «последовательностям», тем не менее, все эти функции могут применяться к любым *итерируемым* объектам, то есть ко всем структурам данных, напоминающим последовательности, по которым можно выполнять итерации. К итерируемым объектам относятся сами последовательности, итераторы, генераторы, ключи словарей, строки в файлах и т. д.

Отображения: словари

Словари являются единственным типом отображений в языке Python. Они являются изменяемыми, неупорядоченными, изменяющимися в размере отображениями ключей на значения и иногда называются хеш-таблицами («хешами»), или ассоциативными массивами. Синтаксис словарей до определенной степени напоминает последовательности, только для доступа к значениям используются не индексы, а ключи, а вместо квадратных (списки) или круглых (кортежи) скобок они определяются с помощью фигурных скобок (`{}`).

Вне всяких сомнений, словари являются одной из важнейших структур данных в языке. Они являются секретной приправой к большинству объектов в языке Python. Независимо от типов объектов и способа их использования высока вероятность, что внутри их заключен словарь, используемый для управления атрибутами объекта. Без долгих разговоров рассмотрим, что из себя представляют словари и как ими пользоваться.

Операции со словарями, методы словарей и функции для работы с отображениями

Ниже приводится несколько примеров использования словарей. Далее мы обсудим, что происходит в этих примерах, а также опишем используемые операторы, методы и функции.

```
>>> book = { 'title': 'Python Web Development', 'year': 2008 }
>>> book
{'year': 2008, 'title': 'Python Web Development'}
>>> 'year' in book
True
>>> 'pub' in book
False
>>> book.get('pub', 'N/A')   # обращение к book['pub'] породило бы ошибку
'N/A'
>>> book['pub'] = 'Addison Wesley'
>>> book.get('pub', 'N/A')   # теперь обращение к book['pub']
                             # не будет ошибкой
'Addison Wesley'
>>> for key in book:
...     print key, ':', book[key]
...
year : 2008
pub : Addison Wesley
title : Python Web Development
```

Ниже кратко описывается, что происходит в этих примерах:

1. Создается словарь, содержащий строку и целое число; оба ключа являются строками.
2. Производится вывод содержимого объекта.
3. Проверяется наличие определенного ключа в словаре (дважды: в первом случае ключ отсутствует, во втором – искомый ключ имеется в словаре).
4. Вызывается метод get для получения значения по указанному ключу (в данном случае возвращается значение по умолчанию).
5. В словарь записывается новая пара ключ-значение.
6. Вторично вызывается тот же метод get, но на этот раз операция завершается успехом.
7. Выполняются итерации по словарю и производится вывод каждой пары ключ-значение.

Теперь забежим немного вперед. В последнем фрагменте кода для обхода ключей словаря использовался цикл for. Это наиболее типичный способ реализации такого обхода. Он также подтверждает утверждение, сделанное выше, о том, что ключи словарей располагаются *не по порядку* (когда необходим определенный порядок следования элементов, следует использовать последовательности!) и такой порядок (точнее, его отсутствие) следования ключей позволяет хешам очень быстро отыскивать значения.

Теперь вернемся к шагу 4 – если ключ, переданный методу get, отсутствует в словаре, возвращается второй аргумент метода (или None, если значение по умолчанию не определено). Получать значения элементов можно также с помощью квадратных скобок, подобно тому, как извле-

Типы данных в платформе Django, подобные словарю

Словарь (или тип данных dict) является встроенным типом данных языка Python, поэтому вполне естественно, что он широко используется в платформе Django. Однако иногда бывает недостаточно способности словаря хранить пары ключ-значение, и требуется поведение, которое словарь не предоставляет. Наиболее ярким примером является объект QueryDict, который хранит параметры запросов GET и POST в объектах HttpRequest. Так как для одного и того же параметра (ключ словаря) допускается передавать несколько значений, а обычные словари такую возможность не поддерживают, потребовалось создать специализированную структуру данных. Если вы работаете с приложением, которое основано на использовании непонятных вам возможностей запросов HTTP, обращайтесь к официальной документации по платформе Django, к разделу «Request and Response Objects».

кается единственный элемент последовательности, d['pub']. Проблема состоит в том, что в данном случае ключ 'pub' еще отсутствует в словаре, поэтому обращение d['pub'] вызовет ошибку. Метод get в этом отношении безопаснее, так как всегда возвращает значение и не вызывает появление ошибки.

Существует еще один, даже более мощный метод с именем setdefault. Он делает то же самое, что и метод get, но при этом, если ему передается несуществующий ключ и значение по умолчанию, он создаст ключ со значением по умолчанию, в результате чего последующие обращения dict.get(key) или dict[key] будут возвращать это значение.

```
>>> d = { 'title': 'Python Web Development', 'year': 2008 }
>>> d.setdefault('pub', 'Addison Wesley')
'Addison Wesley'
>>> d
{'year': 2008, 'pub': 'Addison Wesley', 'title': 'Python Web Development'}
```

Теперь, когда вы уже знаете, как ведет себя хеш, познакомимся с понятием *конфликт ключей*. Этот конфликт возникает, когда выполняется попытка сохранить в словаре разные значения с одним и тем же ключом. В языке Python не распознаются конфликты такого рода, поэтому при попытке присвоить другой объект уже существующему ключу ему просто будет присвоено другое значение, которое затрет прежнее. Ниже демонстрируется, как удалить пару ключ-значение с помощью ключевого слова del, чтобы вернуть словарь в исходное состояние:

```
>>> del d['pub']
>>> d['title'] = 'Python Web Development with Django'
>>> d
{'year': 2008, 'title': 'Python Web Development with Django'}
>>> len(d)
2
```

В первой строке используется инструкция del для удаления пары ключ-значение. Затем существующему ключу title присваивается другая строка, которая замещает значение, существовавшее ранее. Наконец, вызывается универсальная функция len, которая сообщает количество пар ключ-значение в словаре. В табл. 1.5 перечислены некоторые наиболее часто используемые методы словарей.

Таблица 1.5. Популярные методы словарей в языке Python

Метод словаря	Описание
keys	Возвращает ключи словаря (смотрите также метод iterkeys)
values	Возвращает значения словаря (смотрите также метод iter-values)
items	Возвращает пары ключ-значение (смотрите также метод iteritems)
get	Возвращает значение для заданного ключа или значение по умолчанию (смотрите также методы setdefault, fromkeys)
pop	Удаляет ключ из словаря и возвращает значение (смотрите также методы clear, popitem)
update	Дополняет текущий словарь содержимым другого словаря

В заключение о стандартных типах данных

Вы узнали, что из всех стандартных типов данных списки и словари являются наиболее часто используемыми структурами данных в приложениях. Кортежи и словари в первую очередь используются для передачи параметров в функции и возврата значений из функций; использование строк и чисел определяется необходимостью в них. В языке Python имеются и другие типы данных, но мы остановились лишь на тех из них, которые наиболее часто используются в приложениях на платформе Django.

Управление потоком выполнения

Теперь, когда вы в основном познакомились с переменными в языке Python, нам необходимо рассмотреть, что, кроме простого присваивания значений, можно с ними делать. Данные в переменных не будут иметь большого смысла, если к ним не применять логику в форме условных инструкций (позволяющих направить выполнение программы

по разным «путям», в зависимости от складывающихся условий) и циклов (повторяющих выполнение одного и того же фрагмента программного кода, основываясь, как правило, на списках или кортежах того или иного вида).

Условная инструкция

Как и в других языках программирования, в Python имеются инструкции if и else. Инструкция «else-if» в языке Python записывается как elif, как в семействе Bourne языков сценариев командной оболочки (sh, ksh и bash). Условная инструкция в языке Python настолько проста, что принцип ее действия можно продемонстрировать в одном примере.

```
data = raw_input("Enter 'y' or 'n': ")
if data[0] == 'y':
    print "You typed 'y'."
elif data[0] == 'n':
    print "You typed 'n'."
else:
    print 'Invalid key entered!'
```

Циклы

Как и в других языках программирования высокого уровня, в языке Python имеется цикл while. Цикл while продолжает выполнять один и тот же фрагмент программного кода, пока условное выражение возвращает значение True:

```
>>> i = 0
>>> while i < 5:
...     print i
...     i += 1
...
0
1
2
3
4
```

Но, откровенно говоря, следует воздерживаться от использования циклов while, так как в языке Python имеется более мощный механизм циклического выполнения программного кода – цикл for. В других языках программирования цикл for является всего лишь средством выполнения определенного числа циклов, дополняя цикл while. Однако в языке Python цикл for больше похож на цикл foreach, присутствующий в языках сценариев командной оболочки, и больше соответствует стилю языка, предоставляя возможность решать проблему в терминах задачи вместо изобретения подпорок в виде переменных-счетчиков и тому подобного.

```
for line in open('/tmp/some_file.txt'):
    if 'error' in line:
        print line
```

Конечно, если вспомнить генераторы списков, можно обнаружить, что данный цикл легко превращается в генератор списков. Многие простые циклы ничуть не хуже (или даже лучше!) работают в виде генераторов списков. Однако иногда предпочтительнее бывает использовать даже простые циклы for – например, при отладке, когда отсутствует возможность использовать инструкцию print в генераторе списков. Способность верно оценивать, когда лучше использовать генераторы списков, а когда – циклы for, приходит с опытом.

Например, встроенная функция enumerate позволяет одновременно выполнять итерации и вести им счет (последнее невозможно организовать только средствами цикла for), как показано ниже:

```
>>> data = (123, 'abc', 3.14)
>>> for i, value in enumerate(data):
...     print i, value
...
0 123
1 abc
2 3.14
```

Использование функции enumerate в моделях платформы Django

Одним из мест в приложениях на платформе Django, где удобно использовать функцию enumerate, являются определения моделей, особенно при использовании полей в именованном аргументе choices – подробнее о данном именованном аргументе модели рассказывается в главе 4 «Определение и использование моделей». В данном случае использование функции enumerate может выглядеть, как показано ниже:

```
STATUS_CHOICES = enumerate(("solid", "squishy", "liquid"))

class IceCream(models.Model):
    flavor = models.CharField(max_length=50)
    status = models.IntegerField(choices=STATUS_CHOICES)
```

В базе данных значения атрибута status хранятся в виде целых чисел (0, 1, 2), но в интерфейсе администратора Django отображаются текстовые обозначения. Такой подход обеспечивает эффективность хранения данных в базе (если это имеет значение), а также прекрасно подходит для случаев, подобных приведенному, когда сортировка данных по алфавиту не требуется.

Обработка исключений

Подобно другим современным языкам программирования, таким как C++ и Java, язык Python предоставляет механизм обработки исключений. Как мы покажем уже в первом примере этого раздела, он дает программисту возможность определять появление ошибок во время выполнения, открывая путь для выполнения некоторых действий и/или шагов по восстановлению после ошибки и продолжению выполнения программы. Конструкция try-except в языке Python напоминает конструкцию try-catch, имеющуюся в других языках.

Когда в процессе выполнения программы возникает исключение, интерпретатор пытается отыскать подходящий обработчик. Если обработчик не будет найден в текущей функции, интерпретатор передаст исключение выше, в вызывающую функцию, и попытается отыскать обработчик *там*, и т. д. Если исключение достигло верхнего уровня (глобальный уровень программного кода) и обработчик не был найден, то интерпретатор выводит диагностическую информацию, чтобы информировать пользователя об ошибке.

Однако следует иметь в виду, что хотя в большинстве случаев ошибки порождают исключения, тем не менее, исключения не всегда являются результатом ошибки. Иногда они играют роль предупреждений, а иногда могут действовать в качестве сигналов для вызывающих функций, например как сигнал об окончании итераций.

Обработчик исключений может иметь как единственный блок обработки определенного типа исключений, так и состоять из серии блоков, выполняющих обработку различных типов исключений. В нашем первом примере можно видеть, что здесь имеется всего один обработчик (программный код в блоке except):

```
# попытаться открыть файл и в случае ошибки вернуть управление
try:
    f = open(filename, 'r')
except IOError, e:
    return False, str(e)
```

Один и тот же обработчик может обрабатывать несколько типов исключений, для чего достаточно поместить имена исключений в кортеж:

```
try:
    process_some_data()
except (TypeError, ValueError), e:
    print "ERROR: you provide invalid data", e
```

Этот пример обрабатывает исключения двух типов, однако внутрь кортежа можно поместить и большее число исключений.

Можно также создавать несколько обработчиков для нескольких типов исключений.

```
try:
    process_some_data()
except (TypeError, ValueError), e:
    print "ERROR: you provide invalid data", e
except ArithmeticError, e:
    print "ERROR: some math error occurred", e
except Exception, e:
    print "ERROR: you provide invalid data", e
```

В последнем блоке except используется то обстоятельство, что Exception является корневым классом для (почти) всех исключений, поэтому, если возникшее исключение не будет обработано одним из вышестоящих обработчиков, о нем побеспокоится эта последняя инструкция.

Предложение finally

В языке Python также имеется конструкция try-finally. Нас часто волнует не сама обработка ошибок, а обеспечение выполнения программного кода, который *должен* быть выполнен при любых условиях, независимо от того, возникло исключение или нет, – который закроет открытый файл, освободит блокировку, вернет соединение с базой данных обратно в пул и т. д. Например:

```
try:
    get_mutex()
    do_some_stuff()
finally:
    free_mutex()
```

Если никакое исключение не было возбуждено, программный код в блоке finally будет выполнен сразу после завершения блока try. Если же возникнет исключение, блок finally также будет выполнен, но само исключение при этом не будет подавлено и продолжит свое «всплытие» по цепочке вызовов в поисках подходящего обработчика.

Начиная с версии Python 2.5 появилась возможность внедрять предложения except в конструкцию try-finally. (В более ранних версиях это не работает.)

```
try:
    get_mutex()
    do_some_stuff()
except (IndexError, KeyError, AttributeError), e:
    log("ERORR: data retrieval accessing a non-existent element")
finally:
    free_mutex()
```

Возбуждение исключений с помощью инструкции raise

До этого момента мы обсуждали, как перехватывать исключения; теперь зададимся вопросом, как их возбуждать. Для этого можно ис-

пользовать инструкцию raise. Предположим, что была создана некоторая функция, которая требует, чтобы в аргументе ей передавалось положительное целое число больше 0. С применением встроенной функции isinstance, проверяющей тип объекта, реализация этой функции могла бы выглядеть примерно так:

```
def foo(must_be_positive_int):
    """foo() -- принимает положительное целое число и обрабатывает его"""

    # проверить, является ли аргумент целым числом
    if not isinstance(must_be_positive_int, int):
        raise TypeError("ERROR foo(): must pass in an integer!")

    # проверить, является ли число положительным
    if must_be_positive_int < 1:
        raise ValueError("ERROR foo(): integer must be greater than zero!")

    # далее выполняется обычная обработка полученного числа
```

В табл. 1.6 приводится краткий перечень исключений, с которыми вы будете сталкиваться чаще всего во время изучения языка Python.

Таблица 1.6. Наиболее часто возникающие исключения

Исключение	Описание
AssertionError	Проверка в инструкции assert завершилась неудачей.
AttributeError	Была предпринята попытка обратиться к отсутствующему атрибуту, например foo.x, где объект foo не имеет атрибута x.
IOError	Ошибка ввода/вывода; наиболее вероятно, что файл не был открыт.
ImportError	Невозможно импортировать модуль или пакет; наиболее вероятно проблема связана с неправильной настройкой пути поиска.
IndentationError	Синтаксическая ошибка; нарушено правило оформления отступов в программном коде.
IndexError	Попытка использовать индекс, превышающий размер последовательности, например x[5], когда в последовательности x имеется всего три элемента.
KeyError	Попытка обращения к несуществующему в словаре ключу.
KeyboardInterrupt	Была нажата комбинация CTRL-C.
NameError	Попытка использовать переменную, которая еще не была связана с каким-нибудь объектом.
SyntaxError	Программный код не компилируется из-за синтаксической ошибки.
TypeError	Попытка использовать объект, тип которого не соответствует ожидаемому.

Таблица 1.6 (продолжение)

Исключение	Описание
UnboundLocalError	Попытка обратиться к локальной переменной, которой еще не было присвоено значение, вероятно потому, что у вас имеется глобальная переменная с тем же именем и вы считаете, что обращаетесь к *ней*.
ValueError	Передано значение, которое вызывающая программа не ожидает получить, хотя тип значения может быть правильным.

Полный список исключений можно найти в описании модуля exceptions по адресу: *http://docs.python.org/lib/module-exceptions.html*.

Исключения в Django

В платформе Django очень широко используются исключения, как и в любой другой сложной программе на языке Python. По большей части они используются исключительно для внутренних нужд, и вам не придется сталкиваться с ними. Однако некоторые исключения предназначены для прямого использования в приложениях на платформе Django. Например, возбуждение исключения Http404 заставляет платформу Django обработать ошибку HTTP 404 "Not Found" (страница не найдена). Возможность возбуждать исключение Http404 в случае, когда что-то пошло не так, вместо того чтобы возвращать из функции признак ошибки, обеспечивает немаловажное удобство при разработке веб-приложений.

Файлы

Выше уже приводилось несколько примеров программного кода, где использовалась встроенная функция open. Эта функция открывает файлы для чтения или для записи:

```
>>> f = open('test.txt', 'w')
>>> f.write('foo\n')
>>> f.write('bar\n')
>>> f.close()
>>> f = open('test.txt', 'r')
>>> for line in f:
...     print line.rstrip()
...
foo
bar
>>> f.close()
```

Помимо метода write существует метод read, который читает содержимое всего файла и возвращает его в виде единственной строки. В случае текстовых файлов можно использовать метод readlines, который читает строки из файла в список, а похожий метод writelines выводит в файл список строк, которые уже должны завершаться символами перевода строки.

Объект файла сам по себе является итератором, поэтому часто нет необходимости напрямую использовать методы read и readlines, чтобы прочитать его содержимое. Простого цикла for, как в предыдущем примере, вполне достаточно для большинства случаев.

Символы завершения строки (\n, \r\n или \r в зависимости от типа операционной системы) сохраняются, по этой причине в примере вызывается метод rstrip, который удаляет их из прочитанных строк. (В противном случае в выводе появились бы пустые строки, потому что инструкция print автоматически добавляет символ перевода строки.) Точно так же все строки, записываемые в файл методами write и writelines, должны завершаться соответствующим символом перевода строки, чтобы в файле они не оказались объединены в одну строку.

Наконец, имеется еще несколько вспомогательных методов для работы с файлами, которые здесь не рассматриваются, но описание которых легко можно найти в главе «Files and I/O» в книге «Core Python Programming».

Функции

Функции в языке Python создаются очень просто. Выше в этой главе уже приводилось несколько объявлений функций. В этом разделе будут представлены дополнительные аспекты, связанные с функциями, включая следующее (но исчерпывая этим все возможности):

1. Объявление и вызов функций.
2. Именованные аргументы (в вызовах функций).
3. Аргументы со значениями по умолчанию (в объявлениях функций).
4. Функции – это обычные объекты.
5. Анонимные функции и лямбда-функции.
6. Контейнеры с параметрами (в вызовах функций).
7. Переменное число аргументов (в объявлениях функций).
8. Декораторы.

Объявление и вызов функций

Функции объявляются с помощью ключевого слова def, за которым следуют имя функции и список параметров в круглых скобках. (Если функция не имеет параметров, то достаточно просто указать пустую пару скобок.)

```
>>> def foo(x):
...     print x
...
>>> foo(123)
123
```

Как видите, вызов функции выполняется еще проще: достаточно указать имя функции и пару круглых скобок, поместив между ними все необходимые аргументы. Теперь рассмотрим более практичный пример.

```
import httplib
def check_web_server(host, port, path):
    h = httplib.HTTPConnection(host, port)
    h.request('GET', path)
    resp = h.getresponse()
    print 'HTTP Response:'
    print '    status =', resp.status
    print '    reason =', resp.reason
    print 'HTTP Headers:'
    for hdr in resp.getheaders():
        print '    %s: %s' % hdr
```

Что делает эта функция? Она принимает имя хоста или IP-адрес (параметр host) сервера, номер порта (параметр port), путь к странице на сервере (параметр path) и пытается связаться с веб-сервером с указанным именем хоста или IP-адресом и номером порта. Если попытка увенчалась успехом, функция посылает запрос GET, в котором передает указанный путь к странице. Вы можете попробовать вызвать эту функцию, чтобы проверить основной веб-сайт проекта Python, и получить следующий вывод:

```
>>> check_web_server('www.python.org', 80, '/')
HTTP Response:
    status = 200
    reason = OK
HTTP Headers:
    content-length: 16793
    accept-ranges: bytes
    server: Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 mod_ssl/2.2.3 OpenSSL/
0.9.8c
    last-modified: Sun, 27 Apr 2008 00:28:02 GMT
    etag: "6008a-4199-df35c880"
    date: Sun, 27 Apr 2008 08:51:34 GMT
    content-type: text/html
```

Именованные аргументы (в вызовах функций)

В дополнение к «обычным» соглашениям о вызове функций в языке Python имеется возможность использовать *именованные аргументы*, что делает программный код, вызывающий функцию, более понятным, а использование функций более простым. Кроме того, при ис-

пользовании именованных аргументов отпадает необходимость запоминать фиксированный порядок следования параметров. Именованные аргументы записываются в виде key=value, как показано ниже, в модифицированной версии предыдущего примера (вывод опущен для краткости):

```
>>> check_web_server(port=80, path='/', host='www.python.org')
```

При вызове функции интерпретатор Python свяжет объекты с соответствующими переменными, основываясь на указанных именах.

Аргументы со значениями по умолчанию (в объявлениях функций)

Еще одна особенность, которой обладают функции в языке Python, – это возможность задавать значения параметров по умолчанию, что делает необязательным передачу их в виде аргументов. Многие функции

Списки и словари в качестве аргументов со значениями по умолчанию

Мы хотели бы предупредить вас об одной часто встречающейся ошибке, которую допускают пользователи Python. Для этого вернемся к обсуждению отличий при использовании изменяемых и неизменяемых объектов – а именно изменяемых списков и словарей и неизменяемых строк и целых чисел. Вследствие изменяемости списков и словарей их может быть небезопасно использовать в качестве значений по умолчанию аргументов функций, так как они сохраняются между вызовами функций, как показано ниже:

```
>>> def func(arg=[]):
...     arg.append(1)
...     print arg
...
>>> func()
[1]
>>> func()
[1, 1]
>>> func()
[1, 1, 1]
```

Эта особенность изменяемых объектов не является интуитивно понятной, поэтому мы специально о ней здесь упоминаем. Старайтесь учитывать эту особенность; в противном случае вы рискуете однажны обнаружить, что ваши функции ведут себя совсем не так, как ожидалось.

имеют параметры, через которые в каждом вызове передается одно и то же значение, поэтому можно упростить использование таких функций, определив значения по умолчанию.

Значения по умолчанию могут быть связаны с параметрами с помощью знака равенства прямо в объявлении функции. Вернемся к нашему примеру, выполняющему проверку веб-сервера, – большинство веб-серверов принимают запросы от клиентов на порте с номером 80, и для простой проверки «работоспособности веб-сервера» обычно достаточно просто проверить страницу верхнего уровня. Эти значения по умолчанию можно определить так:

```
def check_web_server(host, port=80, path='/'):
```

Не следует путать именованные аргументы и параметры со значениями по умолчанию, так как первые используются только в *вызовах функций*, а вторые – в *объявлениях функций*. Все обязательные параметры должны следовать перед необязательными – их нельзя смешивать или указывать в противоположном порядке.

```
def check_web_server(host, port=80, path): # НЕПРАВИЛЬНО
```

Функции – это обычные объекты

В языке Python функции (и методы) можно воспринимать как обычные объекты и сохранять их в контейнерных объектах, присваивать различным переменным, передавать в виде аргументов другим функциям и т. д. Единственная отличительная черта объектов функций состоит в том, что они могут выполняться, то есть их можно трактовать как функции, добавив круглые скобки с аргументами. Для дальнейшего обсуждения нам необходимо разобраться с тем, что из себя представляют ссылки на объекты.

Ссылки

Когда интерпретатор выполняет инструкцию `def`, он создает объект функции и связывает его с именем, расположенным в текущем пространстве имен. Но это имя может оказаться всего лишь первой ссылкой, или *псевдонимом*, из многих. Каждый раз, когда объект функции передается в вызов другой функции, помещается в контейнер, присваивается локальной переменной и т. д., тем самым создается дополнительная ссылка, или *псевдоним*, на данный объект.

В следующем примере создается не одна, а *две* переменные в глобальном пространстве имен, поскольку после того, как функция будет определена, она становится обычной переменной, как и любая другая:

```
>>> foo = 42
>>> def bar():
...     print "bar"
...
```

Так же, как и в случае любых других объектов, в языке Python можно создать сколько угодно ссылок на одну и ту же функцию. Ниже приводятся несколько примеров, которые поясняют это утверждение. Для начала – обычный случай использования функции bar:

```
>>> bar()
bar
```

Затем выполняется присваивание функции bar другой переменной, с именем baz, после чего к функции, изначально известной под именем bar, можно обращаться по имени baz.

Объекты функций в платформе Django

Тот факт, что объекты функций могут передаваться как любые другие значения, очень часто используется платформой Django. Ниже приводится типичный пример присваивания представлений Django в файлах настройки адресов URL.

```
from django.conf.urls.defaults import *
from myproject.myapp.views import listview

urlpatterns = patterns('',
    url(r'^list/', listview),
)
```

В этом фрагменте в аргументе listview передается сам объект функции, а не строка с ее именем.

Еще одна область, где использование объектов функций дает хороший эффект, – значения по умолчанию для полей модели. Например, если необходимо, чтобы в поле DateField по умолчанию записывалась дата создания, можно передать объект функции из стандартной библиотеки, которая сгенерирует значение при вызове.

```
import datetime

class DiaryEntry(models.Model):
    entry = models.TextField()
    date = models.DateField(default=datetime.date.today)
```

Здесь есть одна хитрость. Если в аргументе default передать datetime.date.today() – обратите внимание на круглые скобки! – функция будет вызвана в момент создания *определения* модели, а это совсем не то, что требуется. Вместо этого в аргументе передается *объект* функции; платформа Django учитывает это и вызовет функцию в момент создания экземпляра класса, благодаря чему будет сгенерировано необходимое значение.

```
>>> baz = bar
>>> bar()
bar
>>> baz()
bar
```

Чтобы вызвать функцию, помещенную в контейнер, достаточно просто сослаться на нее как на любой другой объект, поместив после ссылки круглые скобки и необходимые параметры. Например:

```
>>> function_list = [bar, baz]
>>> for function in function_list:
...     function()
...
bar
bar
```

Обратите внимание, что круглые скобки используются, только когда требуется *вызвать* функцию. Когда функция передается как переменная или объект, следует использовать только ее имя (как это сделано в строке, где создается список function_list). Это подчеркивает различия между ссылкой на *имя* объекта функции – например, bar, и фактическим вызовом, или выполнением ее – например, bar().

Анонимные функции

Анонимные функции – это еще одна особенность функционального стиля программирования на языке Python. Они создаются с помощью ключевого слова lambda и состоят из единственного выражения, значение которого является «возвращаемым значением» функции. Такие функции не объявляются, как обычные функции, и потому не имеют имени, откуда и появился термин *анонимная функция*. Эти функции являются обычными выражениями, которые создаются, используются и уничтожаются обычным способом. Следует отметить разницу между терминами «выражение» и «инструкция», чтобы не путать их в дальнейшем.

Выражения и инструкции

Программный код на языке Python состоит из выражений и инструкций, которые выполняются интерпретатором Python. Основное различие между ними состоит в том, что *выражение* имеет значение – его результатом всегда является объект того или иного типа. Когда выражение вычисляется интерпретатором Python, в результате создается некоторый объект (это может быть любой объект), например 42, 1 + 2, int('123'), range(10) и т. д.

Инструкциями называются строки программного кода, которые *не* имеют результата в виде объектов. Например, инструкции if или print, циклы for и while и т. д. Назначение инструкций очевидно – они

выполняют некоторое действие и ничего не возвращают и ничего не генерируют.

Использование лямбда-функций

Выражение `lambda` имеет следующий синтаксис: `lambda` *аргументы*: *выражение*. После выполнения выражение `lambda` возвращает объект функции, который можно использовать сразу, или сохранить ссылку на него в переменной или передать в виде аргумента, чтобы позднее его можно было использовать как функцию обратного вызова.

Наиболее часто лямбда-функции используются, когда необходимо передать объект функции другой функции, такой как `sorted`, которая,

Лямбда-функции в Django

Лямбда-функции не очень широко используются в платформе Django, но имеется одна область, где они выглядят наиболее удобными: в так называемых декораторах аутентификации, идентифицирующих страницы, которые должны быть доступны только пользователям, обладающим определенными правами. Один из способов реализовать такую возможность состоит в том, чтобы взять объект `User`, представляющий зарегистрировавшегося пользователя, и передать его функции, которая вернет значение `True`, если пользователь обладает необходимыми правами, и `False` – в противном случае.

Такую функцию можно определить с помощью обычной конструкции `def foo():`, но лямбда-функция обеспечивает более компактный способ реализации. Пока у вас нет достаточных знаний, чтобы полностью понять этот пример, но используемые идентификаторы должны быть вам понятны.

```
@user_passes_test(lambda u: u.is_allowed_to_vote)
def vote(request):
    """Обработать голос пользователя"""
```

Первая строка, начинающаяся с символа `@`, является декоратором функции, с которыми вы познакомитесь ниже в этой главе. Декораторы «обертывают» функции (такие, как функция `vote` в этом примере) с целью изменить их поведение. Декоратор `user_passes_test`, встроенная особенность платформы Django, принимает в качестве аргумента любую функцию, принимающую объект `User` и возвращающую логическое значение (`True` или `False`). Поскольку наш тест очень простой – просто возвращаем значение определенного атрибута объекта `User`, вся реализация уместилась в единственную строку.

```
>>> baz = bar
>>> bar()
bar
>>> baz()
bar
```

Чтобы вызвать функцию, помещенную в контейнер, достаточно просто сослаться на нее как на любой другой объект, поместив после ссылки круглые скобки и необходимые параметры. Например:

```
>>> function_list = [bar, baz]
>>> for function in function_list:
...     function()
...
bar
bar
```

Обратите внимание, что круглые скобки используются, только когда требуется *вызвать* функцию. Когда функция передается как переменная или объект, следует использовать только ее имя (как это сделано в строке, где создается список function_list). Это подчеркивает различия между ссылкой на *имя* объекта функции – например, bar, и фактическим вызовом, или выполнением ее – например, bar().

Анонимные функции

Анонимные функции – это еще одна особенность функционального стиля программирования на языке Python. Они создаются с помощью ключевого слова lambda и состоят из единственного выражения, значение которого является «возвращаемым значением» функции. Такие функции не объявляются, как обычные функции, и потому не имеют имени, откуда и появился термин *анонимная функция*. Эти функции являются обычными выражениями, которые создаются, используются и уничтожаются обычным способом. Следует отметить разницу между терминами «выражение» и «инструкция», чтобы не путать их в дальнейшем.

Выражения и инструкции

Программный код на языке Python состоит из выражений и инструкций, которые выполняются интерпретатором Python. Основное различие между ними состоит в том, что *выражение* имеет значение – его результатом всегда является объект того или иного типа. Когда выражение вычисляется интерпретатором Python, в результате создается некоторый объект (это может быть любой объект), например 42, 1 + 2, int('123'), range(10) и т. д.

Инструкциями называются строки программного кода, которые *не* имеют результата в виде объектов. Например, инструкции if или print, циклы for и while и т. д. Назначение инструкций очевидно – они

выполняют некоторое действие и ничего не возвращают и ничего не генерируют.

Использование лямбда-функций

Выражение `lambda` имеет следующий синтаксис: `lambda` *аргументы*: *выражение*. После выполнения выражение `lambda` возвращает объект функции, который можно использовать сразу, или сохранить ссылку на него в переменной или передать в виде аргумента, чтобы позднее его можно было использовать как функцию обратного вызова.

Наиболее часто лямбда-функции используются, когда необходимо передать объект функции другой функции, такой как `sorted`, которая,

Лямбда-функции в Django

Лямбда-функции не очень широко используются в платформе Django, но имеется одна область, где они выглядят наиболее удобными: в так называемых декораторах аутентификации, идентифицирующих страницы, которые должны быть доступны только пользователям, обладающим определенными правами. Один из способов реализовать такую возможность состоит в том, чтобы взять объект `User`, представляющий зарегистрировавшегося пользователя, и передать его функции, которая вернет значение `True`, если пользователь обладает необходимыми правами, и `False` – в противном случае.

Такую функцию можно определить с помощью обычной конструкции `def foo():`, но лямбда-функция обеспечивает более компактный способ реализации. Пока у вас нет достаточных знаний, чтобы полностью понять этот пример, но используемые идентификаторы должны быть вам понятны.

```
@user_passes_test(lambda u: u.is_allowed_to_vote)
def vote(request):
    """Обработать голос пользователя"""
```

Первая строка, начинающаяся с символа @, является декоратором функции, с которыми вы познакомитесь ниже в этой главе. Декораторы «обертывают» функции (такие, как функция `vote` в этом примере) с целью изменить их поведение. Декоратор `user_passes_test`, встроенная особенность платформы Django, принимает в качестве аргумента любую функцию, принимающую объект `User` и возвращающую логическое значение (`True` или `False`). Поскольку наш тест очень простой – просто возвращаем значение определенного атрибута объекта `User`, вся реализация уместилась в единственную строку.

кстати, ожидает получить в аргументе key функцию, применяемую к элементам списка при сортировке для получения желаемого порядка сортировки. Например, если представить, что имеется список сложных объектов, содержащих информацию о людях, и необходимо отсортировать этот список по фамилии (атрибут last_name), это можно реализовать, как показано ниже:

```
sorted(list_of_people, key=lambda person: person.last_name)
```

Этот прием работает потому, что ожидается, что в аргументе key будет передан объект функции, и выражение lambda возвращает анонимную функцию. То же самое можно было бы реализовать немного иначе:

```
def get_last_name(person):
    return person.last_name

sorted(list_of_people, key=get_last_name)
```

В действительности, реализовать то же самое можно даже так:

```
get_last_name = lambda person: person.last_name
sorted(list_of_people, key=get_last_name)
```

Различия между этими тремя вариантами в значительной степени заключаются в удобочитаемости и пригодности к многократному использованию. Первый пример более компактен и при этом достаточно очевидно, благодаря чему является предпочтительным решением проблемы. С другой стороны, многим лямбда-функциям свойственно «перерастать» в обычные функции (например, когда программист замечает, что эта функция могла бы использоваться в разных частях программы), и в этом случае можно было бы подумать об использовании второго варианта.

Третий вариант в действительности далек от практического применения – мы просто хотели показать, что выражение lambda является точным эквивалентом одноразового определения функции, а также подчеркнуть, что функции в языке Python являются обычными объектами.

*args и **kwargs

В этом разделе мы обсудим специальное значение символов * и ** в языке Python, которые связаны с функциями, но обеспечивает различное поведение при использовании в объявлениях и в вызовах функций. Прежде чем двинуться дальше, заметим, специально для программистов C/C++, что символ звездочки не имеет отношения к указателям!

Вообще говоря, если в вызове или в объявлении функции присутствует одиночный символ *, это означает, что подразумевается объект типа tuple (или list), а наличие двух символов ** означает, что подразумевается объект типа dict. Начнем с вызовов функций.

Символы * и ** в вызовах функций

Воспользуемся еще раз функцией check_web_server, которая рассматривалась выше. Ниже еще раз приводится сигнатура функции:

```
def check_web_server(host, port, path):
```

Вызов функции выглядит примерно так: check_web_server('127.0.0.1', 8000, '/admin/'). А что, если вся необходимая информация хранится в виде кортежа? Например:

```
host_info = ('www.python.org', 80, '/') # http://www.python.org/
```

Тогда функцию можно было бы вызвать так:

```
check_web_server(host_info[0], host_info[1], host_info[2])
```

Однако такой метод вызова не отличается ни масштабируемостью (как быть, если функции необходимо передать с десяток аргументов?), ни удобством. Эту проблему можно решить, если использовать одиночный символ *, который выполняет операцию распаковывания, если предшествует кортежу или списку. Следующий фрагмент является точным эквивалентом предыдущей строки программного кода:

```
check_web_server(*host_info)
```

Как видите, это решение выглядит более ясно и элегантно. Использование символов ** со словарем дает аналогичный эффект. Создадим вместо кортежа ('www.python.org', 80, '/') словарь с похожим содержимым:

```
host_info = {'host': 'www.python.org', 'port': 80, 'path': '/'}
```

Функцию можно вызвать так:

```
check_web_server(**host_info)
```

В данном случае будет выполнено распаковывание словаря, каждый ключ которого будет интерпретироваться как имя параметра, а соответствующее ему значение станет аргументом в вызове функции. Другими словами, этот вызов идентичен следующему:

```
check_web_server(host='www.python.org', port=80, path='/')
```

Вы можете использовать любой из этих приемов или оба сразу для вызова функций с позиционными и/или именованными аргументами.

Символы * и ** в сигнатурах функций

Символы * и ** в сигнатурах функций обладают схожим поведением, но играют иную роль: они обеспечивают поддержку *переменного числа аргументов* и позволяют функциям принимать *произвольное число аргументов*.

Если функция определяется с тремя обязательными аргументами (аргументы без значений по умолчанию), вызывающая программа долж-

на передавать функции точно три аргумента. Наличие аргументов со значениями по умолчанию добавляет некоторую гибкость, но и в этом случае вызывающая программа не может передать функции больше аргументов, чем определено в ее объявлении.

Для повышения гибкости можно определить переменное число аргументов, которые будут передаваться функции в виде кортежа, хранящем все передаваемые аргументы, с помощью символа *. Давайте создадим функцию, вычисляющую сумму дневного оборота, как показано ниже:

```
def daily_sales_total(*all_sales):
    total = 0.0
    for each_sale in all_sales:
        total += float(each_sale)
    return total
```

Допустимыми будут все вызовы функции, приведенные ниже:

```
daily_sales_total()
daily_sales_total(10.00)
daily_sales_total(5.00, 1.50, '128.75') # Допускаются не только числа
                                        # с плавающей точкой, но
                                        # и значения любых других типов
```

Совершенно неважно, сколько аргументов будет передано функции, – она обработает их все. Параметр all_sales – это обычный кортеж, который содержит все передаваемые аргументы (именно по этой причине мы можем выполнить обход элементов all_sales в цикле, внутри определения функции).

Имеется также возможность смешивать обычные определения аргументов со списками аргументов переменной длины; в данном случае такой «список аргументов» действует как вместилище для всего, что будет передано, как, например, в гипотетической функции check_web_server, способной принимать дополнительные аргументы.

```
def check_web_server(host, port, path, *args):
```

Примечание

При определении функций с переменным числом аргументов все обязательные аргументы должны быть указаны первыми. Вслед за ними указываются все аргументы со значениями по умолчанию и в самом конце – список аргументов переменной длины.

Точно так же в сигнатурах функций используются символы **, позволяя функции принимать произвольное число именованных аргументов, которые передаются функции в виде словаря.

```
def check_web_server(host, port, path, *args, **kwargs):
```

Теперь функция определена так, что она ожидает получить как минимум три первых аргумента, но с удовольствием примет произвольное число дополнительных позиционных *или* именованных аргументов. Внутри функции теперь можно проверить содержимое кортежа `args` или словаря `kwargs`, и затем использовать или не использовать их содержимое.

Фактически в языке Python существует так называемая универсальная сигнатура метода, которая состоит только из списков аргументов переменной длины.

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

Такую функцию можно вызвать как `f()`, `f(a, b, c)`, `f(a, b, foo=c, bar=d)` и т. д. – она примет все, что ей будет передано. Как в таких функциях обрабатывается содержимое параметров `args` и `kwargs`, конечно, зависит от назначения функции.

Использование **kwargs в классе QuerySets платформы Django: динамическое создание запросов ORM

В функциях платформы Django, выполняющих запросы к базе данных, часто используются именованные аргументы. Например:

```
bob_stories = Story.objects.filter(title__contains="bob",
    subtitle__contains="bob", text__contains="bob",
    byline__contains="bob")
```

Выглядит достаточно очевидно. Ниже показано, как те же именованные аргументы можно передать в виде словаря:

```
bobargs = {'title__contains': 'bob', 'subtitle__contains': 'bob',
    'text__contains': 'bob', 'byline__contains': 'bob'}
bob_stories = Story.objects.filter(**bobargs)
```

А теперь взгляните, как можно было бы создать требуемый словарь динамически:

```
bobargs = dict((f + '__contains', 'bob') for f in ('title',
'subtitle',
    'text', 'byline'))
bob_stories = Story.objects.filter(**bobargs)
```

Как видите, *этот* прием может с успехом использоваться для упрощения избыточных запросов или, в еще более общем случае, для сборки аргументов, имена которых определяются динамически (например, исходя из значений параметров в форме поиска).

Декораторы

Последняя концепция, которая кажется, пожалуй, самой трудной для понимания при изучении функций и функционального программирования на языке Python, – это *декораторы*. Декораторы в языке Python представляют собой механизм, позволяющий изменять, или «декорировать», поведение функций, заставляя их работать несколько иначе, чем предполагалось с самого начала, или производить какие-либо дополнительные действия. Если хотите, декоратор – это «обертка вокруг функции». В числе таких дополнительных действий можно назвать регистрацию событий (журналирование), согласование по времени, фильтрацию и т. д.

В языке Python обернутая или декорированная функция (объект) опять присваивается своему первоначальному имени, чтобы сохранить совместимость с первоначальной версией; по этой причине использование декораторов можно сравнить с «наложением» дополнительных функциональных возможностей поверх существующих.

В простейшем случае синтаксис применения декоратора выглядит, как показано ниже:

```
@deco
def foo():
    pass
```

В этом примере deco – это функция-декоратор, которая «декорирует» функцию foo. Она принимает функцию foo, добавляет к ней некоторую функциональность и затем присваивает получившуюся функцию имени foo. Синтаксис @deco идентичен следующей строке программного кода (учитывая, что foo – это существующий объект функции):

```
foo = deco(foo)
```

Следующий простой пример демонстрирует, как реализовать регистрацию каждого вызова функции:

```
def log(func):
    def wrappedFunc():
        print "*** %s() called" % func.__name__
        return func()
    return wrappedFunc

@log
def foo():
    print "inside foo()"
```

Теперь, если выполнить этот фрагмент программного кода, мы будем получать следующий вывод:

```
>>> foo()
*** foo() called
inside foo()
```

Выше в этой главе мы видели пример декоратора, который принимает аргумент.

```
@user_passes_test(lambda u: u.is_allowed_to_vote)
```

В данном случае мы вызываем функцию, возвращающую фактический декоратор – функция user_passes_test сама по себе не является декоратором, но это функция, которая принимает аргументы, использует их и возвращает готовый к использованию декоратор. Синтаксис вызова такой функции выглядит примерно так, как показано ниже:

```
@decomaker(deco_args)
def foo():
    pass
```

Эта конструкция эквивалентна следующему фрагменту (не забывайте, что в языке Python допускается объединять выражения в цепочки):

```
foo = decomaker(deco_args)(foo)
```

Здесь функция, создающая декоратор («decorator-maker», или decomaker), принимает аргумент deco_args и возвращает декоратор, который принимает и обертывает функцию foo.

Следующий заключительный пример демонстрирует возможность применения сразу нескольких декораторов:

```
@deco1(deco_args)
@deco2
def foo():
    pass
```

Мы не будем обсуждать этот фрагмент, но, исходя из примеров выше, можно заключить, что он идентичен следующему фрагменту:

```
foo = deco1(deco_args)(deco2(foo))
```

Вы все еще можете задаваться вопросом: «Зачем нужны декораторы? В чем их особенность?». Откровенно говоря, обертывание функций – не новинка в языке Python и нет ничего особенного в том, чтобы взять объект, модифицировать его и присвоить обратно той же переменной. Но особенным является то, что декораторы позволяют упростить синтаксис, добвив единственный символ @.

Более подробное и понятное описание декораторов приводится в руководстве Кента Джона (Kent John) «Python Decorators» по адресу *http://personalpages.tds.net/~kent37/kk/00001.html*.

Объектно-ориентированное программирование

Прежде всего следует отметить, что этот раздел не является учебным пособием по объектно-ориентированному программированию (ООП). Мы лишь собираемся немного углубиться в проблему создания и ис-

пользования классов на языке Python. Если вы не знакомы с ООП, то язык Python даст вам самый простой способ изучить его; тем не менее, будет лучше, если для начала вы прочитаете специализированное учебное пособие, хотя это и не обязательно. Главная цель ООП состоит в том, чтобы обеспечить логическое отображение объектов реального мира в программном коде, а также обеспечить возможность многократного и совместного использования программного кода. Кроме того, мы познакомим вас с некоторыми особенностями, характерными только для языка Python.

Определение классов

В нашем первом примере мы смоделируем объект реального мира – создадим адресную книгу. Чтобы создать класс на языке Python, необходимо использовать ключевое слово class, указать имя нового класса и одно или более имен *базовых классов*, или классов, на основе которых создается новый класс. Например, если вы хотите создать класс Car[1] или Truck[2], в качестве базового можно использовать класс Vehicle[3]. Если отсутствуют классы, которые можно было бы использовать в качестве базовых, просто используйте корневой класс, или тип данных, – object, как это делается в примере класса AddressBookEntry, представляющего запись в адресной книге:

```
class AddressBookEntry(object):
    version = 0.1

    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

    def update_phone(self, phone):
        self.phone = phone
```

Статические члены класса, такие как version, можно создавать простой инструкцией присваивания внутри определения класса, – эти члены представляют собой обычные переменные, принадлежащие классу и совместно используемые всеми экземплярами класса. Методы определяются как обычные функции, в которых первым параметром является ссылка self на сам объект – язык Python требует явного ее объявления.

Переменная self ссылается на конкретный экземпляр класса (в других языках программирования аналогичная ссылка называется не self, а this). Если определение класса можно уподобить чертежу, то экземпляр класса – это воплощение класса, реальный объект, которым можно управлять в процессе выполнения программы. Форма за-

---

[1] Легковой автомобиль. – *Прим. перев.*

[2] Грузовик. – *Прим. перев.*

[3] Транспортное средство. – *Прим. перев.*

писи имен переменных, квалифицированных словом self с точкой, указывает на принадлежность атрибута экземпляру, то есть принадлежность данного объекта конкретному экземпляру. Если представить, что переменная name является атрибутом экземпляра, тогда необходимо использовать полностью квалифицированное имя self.name, чтобы обратиться к ней.

Для тех из вас, кто пришел из других объектно-ориентированных языков, заметим, что в языке Python отсутствуют понятия конструктора и деструктора – в языке нет ключевых слов new и free. Более подробно мы поговорим об этом чуть ниже.

Создание экземпляров

В других языках программирования новые экземпляры создаются с помощью инструкции new, но в языке Python достаточно просто вызвать класс, как если бы он был простой функцией. Вместо «конструктора» классы в языке Python имеют метод инициализации с именем __init__. Когда создается экземпляр класса, ему необходимо передать все аргументы, необходимые методу __init__. Когда интерпретатор создает объект, он автоматически вызывает метод __init__ с указанными аргументами, после чего вручает вам вновь созданный объект.

Что касается вызова методов, можно смело утверждать, что делается это так же просто, как вызов обычных функций. Несмотря на то, что в объявлении методов обязательно должен присутствовать параметр self, Python дает возможность не задумываться об этом при вызове методов (обычным *связанным* способом), автоматически передавая аргумент self *без* вашего участия. Ниже приводятся два примера создания экземпляров класса AddressBookEntry:

```
john = AddressBookEntry('John Doe', '408-555-1212')
jane = AddressBookEntry('Jane Doe', '650-555-1212')
```

Не забывайте, что при создании каждого экземпляра интерпретатор вызывает метод __init__ и затем возвращает объект. Взгляните еще раз – в вызовах отсутствует аргумент self, передаются только имя и номер телефона. Ссылка self передается интерпретатором автоматически, без вашего участия.

После создания экземпляров появляется возможность обращаться к их атрибутам напрямую, например, john.name, john.phone, jane.name, jane.phone. Как видно в следующем примере, мы можем обращаться к атрибутам экземпляров без всяких ограничений:

```
>>> john = AddressBookEntry('John', '408-555-1212')
>>> john.phone
'408-555-1212'
>>> john.update_phone('510-555-1212')
>>> john.phone
'510-555-1212'
```

Еще раз обратите внимание на то, что в сигнатуре метода update_phone указано два параметра, self и newphone, но от нас требуется передать только новый номер телефона, а экземпляр объекта, на который ссылается переменная john, в параметре self передает интерпретатор.

Язык Python поддерживает также **динамические атрибуты экземпляра**, то есть атрибуты, которые не объявляются в определении класса, но которые могут добавляться к экземпляру в процессе выполнения программы.

```
>>> john.tattoo = 'Mom'
```

Это определенно выгодная особенность, демонстрирующая гибкость языка Python. Такие атрибуты могут создаваться когда угодно, и таких атрибутов можно создать сколько угодно.

Создание подклассов

Создание подкласса сродни созданию класса, только в этом случае необходимо указать один или более базовых классов (вместо корневого класса object). Продолжая предыдущий пример, создадим класс записей в адресной книге с информацией о заказчике.

```
class EmployeeAddressBookEntry(AddressBookEntry):
    def __init__(self, name, phone, id, social):
        AddressBookEntry.__init__(self, name, phone)
        self.empid = id
        self.ssn = social
```

Обратите внимание на отсутствие инструкций, присваивающих значения аргументов phone и name атрибутам self.phone и self.name – эти присваивания будут выполнены вызовом метода AddressBookEntry.__init__. Когда в дочернем классе переопределяется метод базового класса, как в данном примере, необходимо явно вызывать его (оригинальный метод базового класса), что мы и сделали. Обратите внимание, что на этот раз мы явно передаем аргумент self, потому что вместо *экземпляра* мы использовали сам *класс* AddressBookEntry.

В любом случае помимо потенциальной возможности переопределять методы базового класса подклассы наследуют все атрибуты и методы базового класса, то есть наш класс EmployeeAddressBookEntry имеет атрибуты name и phone, а также метод update_phone. Платформа Django, так же, как и большинство других программ и платформ, написанных на языке Python, широко использует механизм наследования как для своих собственных нужд, так и для предоставления функциональных возможностей в распоряжение пользователей платформы, заинтересованных в этом.

Вложенные классы

Так же, как для создания декораторов можно использовать «вложенные функции», внутри определений классов можно создавать *вложенные классы*, например:

```
class MyClass(object):
    class InnerClass:
        pass
```

Этот вложенный класс является самым обычным классом, но он видим только экземплярам класса MyClass. Эта особенность языка Python относится к разряду малоизвестных, но она имеет приложение в платформе Django (смотрите врезку «Классы и модели Django» ниже). Единственный, но очень важный вложенный класс, который вам наверняка придется использовать при разработке приложений на платформе Django, – это вложенный класс Meta.

Классы и модели Django

Модели данных платформы Django и основа большинства Django-приложений – это классы, наследующие встроенный класс платформы Django django.db.models.Model. Класс Model обладает широчайшими возможностями, и мы будем рассматривать их в главе 4. А пока взгляните на фрагмент программного кода из приложения, которое мы будем создавать в главе 2 «Django для нетерпеливых: создание блога»:

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

    class Meta:
        ordering = ('-timestamp',)
```

Здесь определяется новый класс BlogPost, наследующий класс django.db.models.Model и добавляющий три дополнительных поля. Благодаря механизму наследования класс BlogPost также получает другие атрибуты и методы класса Model, среди которых не последнее место занимают методы, позволяющие запрашивать объекты класса BlogPost из базы данных, создавать новые объекты и обращаться к связанным элементам.

Регулярные выражения

Наряду с другими приемами платформа Django использует прием сопоставления со строковыми шаблонами, известными как **регулярные выражения**, для определения адресов URL в пределах веб-сайта. Без использования регулярных выражений пришлось бы определять все возможные адреса URL – этот способ неплохо подходит, например, для адресов */index/* или */blog/posts/new/*, но никуда не годится для адресов, создаваемых динамически, таких как */blog/posts/2008/05/21/*.

Во многих книгах и в электронной документации дается достаточно подробное описание, что такое регулярные выражения и как они создаются, поэтому здесь мы не будем тратить на это слишком много времени. Хорошую подборку ресурсов с информацией по этому вопросу можно найти на сайте *withdjango.com*. Остальная часть этого раздела предполагает, что вы уже знакомы с регулярными выражениями (или познакомитесь прямо сейчас), поэтому в ней мы будем рассматривать вопросы, связанные с их использованием в языке Python.

Модуль re

Механизмы для работы с регулярными выражениями доступны в языке Python в виде модуля `re`. Одним из часто используемых компонентов модуля является функция `search`. Функция `re.search` возвращает объект совпадения, обладающий методами `group` и `groups`, которые могут использоваться для извлечения совпадений.

```
>>> import re
>>> m = re.search(r'foo', 'seafood')
>>> print m
<_sre.SRE_Match object at ...>
>>> m.group()
'foo'
>>> m = re.search(r'bar', 'seafood')
>>> print m
None
```

В случае успеха функция `search` возвращает объект класса `Match`, обладающий методом `group`, который может вызываться для получения строки совпадения. В случае отсутствия совпадений функция возвращает значение `None`. Обратите внимание на использование спецификатора `r''` «сырых» строк – как уже упоминалось выше в этой главе, для определения регулярных выражений предпочтительнее использовать «сырые» строки, так как в этом случае отпадает необходимость экранировать специальные символы, такие как символ обратного слеша.

Поиск и соответствие

Нам необходимо понять разницу между процедурой *поиска*, в ходе которой выявляются совпадения с шаблоном, которые могут находиться

в любом месте проверяемой строки, и процедурой выявления *соответствия*, когда проверяется соответствие всей строки заданному шаблону. Например:

```
>>> import re
>>> m = re.match(r'foo', 'seafood')
>>> if m is not None: print m.group()
...
>>> print m
None
>>> m = re.search(r'foo', 'seafood')
>>> if m is not None: print m.group()
...
'foo'
```

В первом примере функция `re.match` вернула пустое значение, или `None`, потому что регулярное выражение `r'foo'` соответствует только части строки `'seafood'`. Функция `re.search` вернула определенный результат, потому что она не требует, чтобы строка полностью совпадала с шаблоном.

Типичные ошибки

В этом разделе мы обсудим некоторые из проблем, с которыми приходится сталкиваться начинающим программистам на языке Python, например: Как создать кортеж с одним элементом? Или, почему повсюду в объектно-ориентированном программном коде на языке Python я вижу имя `self`?

Кортежи с единственным элементом

Для начинающего программиста достаточно очевидно, что `()` и `(123, 'xyz', 3.14)` являются кортежами, но совсем не очевидно, что `(1)` не является таковым. Круглые скобки в языке Python имеют несколько значений. В выражениях скобки применяются для группировки. Кортеж из одного элемента в языке Python создается с помощью пусть и некрасивого, но необходимого приема, который заключается в том, чтобы добавить запятую после единственного элемента: `(1.)`.

Модули

Выше уже было показано два различных способа импортирования модулей и их атрибутов:

```
import random
print random.choice(range(10))
```

И

```
from random import choice
print choice(range(10))
```

В первом примере используется прием импортирования пространства имен, когда имя модуля устанавливается, как глобальная переменная, и появляется возможность получить доступ к функции choice как к атрибуту этой глобальной переменной. Во втором примере имя choice импортируется непосредственно в глобальное пространство имен (имя модуля при этом не импортируется). Благодаря этому отпадает необходимость обращаться к нему, как к атрибуту модуля. В этом случае мы получаем только имя самого атрибута.

Среди начинающих программистов на языке Python бытует мнение, что при использовании второго способа модуль целиком не импортируется, а импортируется только функция. Это не так. В действительности загружается весь модуль, но сохраняется только ссылка на эту функцию. Конструкция from-import не дает никаких преимуществ в смысле экономии памяти или производительности.

Возможно ли импортировать модуль больше чем один раз?

Часто источником беспокойств для начинающих программистов является ситуация, когда имеются модули m.py и n.py, и оба они импортируют модуль foo.py. Будет ли модуль foo импортирован дважды, если из модуля m импортировать модуль n? Если ответить коротко – да, но это не совсем то, о чем вы подумали.

В языке Python есть понятие импортирования модуля и есть понятие загрузки модуля. Модуль может импортироваться много раз, но загруженным в память он может быть только единожды. Проще говоря, когда интерпретатор Python выполняет инструкцию import, импортирующую модуль, который уже был загружен, он пропускает процедуру загрузки, поэтому вам нет нужды беспокоиться, что повторное импортирование модуля приведет к напрасному расходованию памяти.

Пакеты

Пакеты в языке Python обеспечивают способ распределения набора модулей по каталогам файловой системы и использование *точечной* нотации для доступа к модулям в подпакетах, как если бы они были обычными атрибутами другого объекта. Другими словами, если вы создаете программный продукт, содержащий сотню модулей, не очень осмысленно помещать их все в общий каталог. Это возможно, но нужно ли? Почему бы не опереться на файловую систему для организации этих модулей в логически связанную структуру?

Например, допустим, что у нас имеется приложение, обеспечивающее возможности телефонии. Для него можно было бы организовать структуру каталогов, которая выглядит, как показано ниже:

```
Phone/
    __init__.py
    util.py
    Voicedata/
```

```
        __init__.py
        Pots.py
        Isdn.py
    Fax/
        __init__.py
        G3.py
    Mobile/
        __init__.py
        Analog.py
        Digital.py
```

Phone – это каталог верхнего уровня, или пакет. В нем находятся подпакеты, которые в действительности являются подкаталогами, содержащими другие модули Python. Вы наверняка обратили внимание на файлы с именем __init__.py, которые имеются в каждом подкаталоге. Эти файлы сообщают интерпретатору Python, что файлы в этих подкаталогах должны восприниматься как подпакеты, а не как простые файлы. Файлы __init__.py, обычно пустые, могут содержать программный код инициализации, который требуется выполнить прежде, чем можно будет использовать какой-либо программный код в подпакетах.

Допустим, что нам требуется вызвать функцию dial, которая обслуживает аналоговые сотовые телефоны. Для этого можно было бы выполнить следующий фрагмент:

```
import Phone.Mobile.Analog
Phone.Mobile.Analog.dial()
```

Такой способ выглядит немного громоздким. Для повседневной работы вы наверняка предпочли бы использовать какие-нибудь сокращения – по причинам, связанным с производительностью, о чем уже говорилось выше, но главным образом, чтобы уберечь себя от необходимости слишком много вводить с клавиатуры! В языке Python и в платформе Django есть все необходимое для обеспечения простоты. Ниже показано, какой программный код вы, вероятно, увидели бы в готовом приложении:

```
import Phone.Mobile.Analog as pma
pma.dial()
```

Изменяемость

Программисты, плохо знакомые с языком Python, часто спрашивают, как передаются параметры функций – «по ссылке» или «по значению»? Ответ на этот вопрос не так прост и сводится к тому, что «это зависит» от того, какие объекты передаются. Некоторые объекты передаются в функции по ссылке, а другие копируются, или передаются по значению. Наблюдаемое поведение зависит от способности объекта изменяться, которая в свою очередь определяется его типом. Вследствие такой двойственности программисты на языке Python вообще не упо-

требляют термины «по ссылке» и «по значению»; вместо этого они выясняют, является ли объект изменяемым или неизменяемым.

Изменяемость – это способность объекта изменять свое значение. В языке Python любой объект имеет три атрибута: тип, числовой идентификатор и значение. Что такое тип и значение объекта, на наш взгляд, должно быть очевидно. Числовой идентификатор – это просто некоторое число, которое уникальным образом идентифицирует объект, позволяя интерпретатору отличить его от любых других объектов.

Все три атрибута практически всегда доступны только для чтения, то есть они не могут изменяться в течение всего времени существования объекта. Единственное возможное исключение – значение: если значение может изменяться, значит, объект изменяемый, в противном случае – неизменяемый.

Простые «скалярные» типы данных, такие как целые числа и другие числовые типы, являются неизменяемыми. Точно так же неизменяемыми являются строковые типы, такие как `str` и `unicode`, а также кортежи. Все остальные типы – списки, словари, классы, экземпляры классов и т. д. – являются изменяемыми.

Примечание

Как видите, способность изменяться – это очень важный аспект программирования на языке Python. Это объясняет, почему в языке Python предлагается два типа «списков», `list` и `tuple`, из которых первый тип является изменяемым, а второй – нет. Возможность иметь объект, напоминающий список, для использования в случаях, когда требуются неизменяемые объекты, например, в качестве ключей словаря, может оказаться весьма полезной.

Какое влияние оказывает способность изменяться на вызовы методов

Одна из проблем, обусловленных способностью объектов изменяться, связана с вызовом методов этих объектов. Если вызываемый метод модифицирует изменяемый объект каким-либо способом, он выполняет изменения непосредственно в самом объекте, то есть, вместо того чтобы вернуть измененную копию, метод модифицирует саму структуру данных (в таких случаях методы возвращают значение `None`).

Типичным примером таких методов может служить метод `list.sort`, выполняющий сортировку самого списка, вместо того чтобы вернуть отсортированную копию. Подобное поведение сбивает с толку многих начинающих программистов, не ожидающих этого! Многие другие методы списков, такие как `reverse` и `extend`, а также некоторые методы словарей, такие как `update` (добавляет в словарь новые пары ключ-значение), тоже изменяют сам объект.

К счастью, начиная с версии 2.4, язык Python предлагает встроенные функции, такие как `sorted` и `reversed`, принимающие итерируемые

объекты и возвращающие их копии, отсортированные в прямом или обратном порядке. Эти функции удобно использовать в случаях, когда нежелательно изменять сами объекты или когда появляется желание сэкономить несколько инструкций программного кода. Если так получилось, что вы пользуетесь версией Python 2.3, то, чтобы получить модифицированную копию списка, вам придется вручную скопировать список (обычно с помощью инструкции `newlist = list(mylist)` или `newlist = mylist[:]`) и затем вызвать метод `sort` копии объекта.

Копирование объектов и изменяемость

Рассмотрим наиболее типичные ошибки, совершаемые начинающими программистами, которые связаны с изменяемостью и с копированием объектов. Вспомните, как в начале раздела говорилось, что неизменяемые объекты передаются по значению, а изменяемые – по ссылке. Это справедливо не только для случаев передачи аргументов функциям, но и для других разновидностей «копирования» объектов: неизменяемые объекты, такие как целые числа, – действительно копируются, но в случае копирования изменяемого объекта копируется только *ссылка* на него, что в результате дает нам один объект в памяти и две ссылки на него.

Чтобы понять, почему это так важно, рассмотрим следующий пример, где имеется список, вложенный в другой список:

```
>>> mylist = [1, 'a', ['foo', 'bar']]
>>> mylist2 = list(mylist)
>>> mylist2[0] = 2
>>> mylist[2][0] = 'biz'
```

Вы могли бы ожидать, что изменение списка, вложенного в список `mylist`, коснется только самого списка `mylist` и не затронет список `mylist2`, но это не так! Взгляните на новые значения обоих списков.

```
>>> print mylist
[1, 'a', ['biz', 'bar']]
>>> print mylist2
[2, 'a', ['biz', 'bar']]
```

Первые два объекта в списке `mylist` – целое число и строка – являются неизменяемыми, поэтому список `mylist2` получил свой собственный целочисленный и строковый объекты – вследствие чего операция изменения 1 на 2 дала ожидаемый результат. Однако третьим объектом в списке `mylist` является список, то есть изменяемый объект, и поэтому в список `mylist2` была скопирована лишь ссылка на него. Так как третьим объектом в обоих списках является ссылка на один и тот же объект в памяти, любые изменения в нем, произведенные с помощью одного из родительских списков, будут отражаться на другом.

Этот вид копирования, который мы только что видели, называется **поверхностным копированием**, потому что при этом копируются ссылки

на изменяемые объекты, а не их значения. Если действительно потребуется скопировать значения изменяемых объектов, то есть выполнить **глубокое копирование**, необходимо импортировать модуль copy и использовать функцию copy.deepcopy. Прежде чем использовать эту функцию, внимательно прочитайте ее описание – такой способ копирования носит по своей природе рекурсивный характер (могут возникнуть проблемы, если в копируемом объекте имеются циклические ссылки!) и не все объекты доступны для глубокого копирования.

Конструктор и метод инициализации

Язык Python является объектно-ориентированным, но он имеет несколько отличий от традиционных объектно-ориентированных языков, одним из которых является отсутствие явного понятия конструктора. В языке Python отсутствует ключевое слово new, поэтому вам не придется создавать экземпляры классов вручную. Интерпретатор сам создаст экземпляр и автоматически вызовет метод инициализации; это первый метод, который вызывается сразу после создания объекта в памяти, но непосредственно перед тем, как интерпретатор вернет его вам. Этот метод всегда называется __init__.

Чтобы создать экземпляр класса или, говоря другими словами, создать объект, необходимо вызвать сам класс, как если бы он был функцией.

```
>>> class MyClass(object):
...     pass
...
>>> m = MyClass()
```

Кроме того, так как интерпретатор автоматически вызывает метод __init__, то, если он принимает какие-либо параметры, вы должны передать их в «вызов» класса.

```
>>> from time import ctime
>>> class MyClass(object):
...     def __init__(self, date):
...         print "instance created at:", date
...
>>> m = MyClass(ctime())
instance created at: Wed Aug 1 00:59:14 2007
```

Точно так же программисты на языке Python обычно не реализуют деструкторы и не уничтожают свои объекты, а вместо этого они просто позволяют объектам покинуть текущую область видимости (в этот момент они будут подвергнуты утилизации механизмом сборки мусора). Тем не менее, объекты в языке Python могут определять метод __del__, который по своему назначению напоминает деструкторы в других языках программирования, а кроме того, существует возможность явного уничтожения объектов с помощью инструкции del (например, del my_object).

Динамические атрибуты экземпляра

Еще один источник непонимания для начинающих программистов (имеющих опыт работы с другими объектно-ориентированными языками программирования) заключается в возможности динамически добавлять атрибуты экземпляра уже после определения класса и создания экземпляра. Возьмем для примера класс AddressBook:

```
>>> class AddressBook(object):
...     def __init__(self, name, phone):
...         self.name = name
...         self.phone = phone
...
>>> john = AddressBook('John Doe', '415-555-1212')
>>> jane = AddressBook('Jane Doe', '408-555-1212')
>>> print john.name
John Doe
>>> john.tattoo = 'Mom'
>>> print john.tattoo
Mom
```

Обратите внимание: атрибут self.tattoo не упоминается ни в объявлении класса, ни в объявлениях методов класса, его нет даже в методе __init__! Мы создали этот атрибут динамически, во время выполнения. Такие атрибуты называются *динамическими атрибутами экземпляра*.

Стиль оформления программного кода (PEP 8 и не только)

Существует масса предложений и рекомендаций по «надлежащему оформлению программного кода», но главное их требование сводится к тому, чтобы придать исходному программному коду «питонический» вид. Программные системы, следующие положениям философии языка Python, таким как быть проще, не повторяться, создавать удобочитаемый программный код, находить изящные решения, многократно использовать программный код и т. д., придерживаются этого вида, и платформа Django не исключение. В нашем ограниченном пространстве мы можем привести лишь несколько основных рекомендаций. Остальные придут к вам по мере накопления опыта общения с языком Python, платформой Django и с их очень благожелательными сообществами. В документе PEP 8 (*http://www.python.org/dev/peps/pep-0008*) вы найдете официальное руководство по оформлению программного кода.

Отступы в четыре пробела

В языке программирования, где границы блоков определяются отступами, а контингент пользователей чрезвычайно разнообразен, ни у ко-

го не вызывает сомнений, что редактировать исходный программный код на языке Python очень утомительно для глаз, если отступы оформляются одним или двумя пробелами. Если оформлять отступы восемью пробелами, легко можно выйти за пределы экрана. Неустаревшая рекомендация Гвидо (Guido), приведенная в его оригинальном эссе, – оформлять отступы четырьмя пробелами, – золотая середина!

Используйте пробелы, но не символы табуляции

Независимо от платформы, на которой ведется разработка, всегда есть вероятность, что ваш программный код будет перемещен или скопирован на другую машину с другой архитектурой или будет выполняться в другой операционной системе. Поскольку на разных платформах символы табуляции интерпретируются по-разному – например, на платформе Win32 один символ табуляции считается равным четырем пробелам, в любых системах UNIX, а также POSIX-совместимых системах, один символ табуляции считается равным восьми пробелам, то резонно будет вообще воздержаться от использования символов табуляции.

Если вам доведется столкнуться с ситуацией, когда интерпретатор Python сообщит, что программный код содержит синтаксическую ошибку, тогда как на экране все выглядит правильно, велика вероятность, что где-то закрался символ табуляции, потому что текстовый редактор показывает вам «ложное» представление того, как в действительности выглядит программный код. Выполнив явное преобразование всех символов табуляции в пробелы, вы увидите, где произошел сдвиг.

Не записывайте короткие блоки программного кода в одной строке с заголовком инструкции

Имеется возможность записывать составные инструкции в одну строку, например:

```
if is_finished(): return
```

Однако мы рекомендуем разбивать их на несколько строк, например так:

```
if is_finished():
    return
```

Основная причина этой рекомендации заключается в обеспечении удобочитаемости программного кода, а кроме того, вам не придется выполнять лишнюю правку, когда потребуется добавить в блок несколько новых строк.

Создавайте строки документирования

Документирование программного кода может оказаться очень полезным, а язык Python обеспечивает не только возможность его докумен-

тирования, но и доступ к описанию во время выполнения. Строки документирования могут создаваться для модулей, классов, а также для функций и методов. Ниже в качестве примера приводится простой сценарий foo.py:

```
#!/usr/bin/env python
"""foo.py -- пример модуля, демонстрирующий строки документирования"""

class Foo(object):
    """Foo() - пустой класс ... для последующей разработки"""

def bar(x):
    """bar(x) - строка документирования функции bar, выводит аргумент 'x'"""
    print x
```

Когда мы сказали «доступ во время выполнения», мы имели в виду следующее: если запустить интерактивный сеанс работы с интерпретатором и импортировать ваш модуль, вы сможете получить доступ к описанию, обратившись к атрибуту __doc__ модуля, класса или функции.

```
>>> import foo
>>> foo.__doc__
'foo.py -- пример модуля, демонстрирующий строки документирования'
>>> foo.Foo.__doc__
'Foo() - пустой класс ... для последующей разработки'
>>> foo.bar.__doc__
'bar(x) - строка документирования функции bar, выводит аргумент "x"'
```

Кроме того, для вывода строк документирования в форматированном виде можно использовать встроенную функцию help. Мы приведем лишь один пример – для модуля (но вы сможете сделать то же самое для класса или для функции).

```
>>> help(foo)
Help on module foo:
(Справка по модулю foo:)

NAME
(ИМЯ)
    foo - foo.py -- пример модуля, демонстрирующий строки документирования

FILE
(ФАЙЛ)
    c:\python25\lib\site-packages\foo.py

CLASSES
(КЛАССЫ)
    __builtin__.object
        Foo

    class Foo(__builtin__.object)
    | Foo() - пустой класс ... для последующей разработки
    |
    | Data descriptors defined here:
    | (Здесь определяются описатели данных:)
```

```
 |
 | __dict__
 |     dictionary for instance variables (if defined)
 |     (словарь переменных экземпляра (если определен))
 |
 | __weakref__
 |     list of weak references to the object (if defined)
 |     (список "слабых" ссылок на объект (если определен)

FUNCTIONS
(ФУНКЦИИ)
    bar(x)
        bar(x) - строка документирования функции bar, выводит аргумент "x"
```

Для большинства программных компонентов, входящих в состав стандартной библиотеки языка Python, уже имеются строки документирования. Поэтому, когда потребуется справочная информация о встроенных функциях, методах встроенных типов или любых атрибутах модулей и пакетов, вы легко сможете получить ее с помощью функции `help` – например, `help(open)`, `help(''.strip)`, `help(time.ctime)` и т. д. При этом предполагается, что все необходимые модули были импортированы.

В заключение

В этой объемной вводной главе мы попытались представить язык Python новым разработчикам приложений на платформе Django. Очевидно, что мы не могли дать всестороннего описания языка, как хотелось бы, но не забывайте, что эта книга рассказывает о платформе Django, а не о языке Python. Тем не менее, мы постарались вместить как можно больше сведений о Python, которые потребуются для успешной работы с Django.

В первых трех четвертях главы в основном рассказывалось о языке Python и давались некоторые пояснения, имеющие отношение к Django. В последней части главы рассказывалось о типичных ошибках, давались рекомендации по оформлению программного кода и т. д. Хотелось бы надеяться, что нам удалось дать достаточный объем сведений о языке, чтобы вы смогли читать и писать несложный программный код на языке Python и войти в мир Django.

В следующей главе мы быстро введем вас в мир Django и менее чем за 20 минут создадим приложение блога. Этот блог не такой функциональный, как коммерческие версии, но на его примере вы получите общее представление о том, насколько быстро можно создавать приложения на платформе Django, и одновременно потренируетесь в применении навыков программирования на языке Python, обретенных в данной главе.

2

Django для нетерпеливых: создание блога

Платформа Django позиционируется как «веб-платформа для стремящихся к совершенству, достижимому в реальные сроки». Поэтому не будем терять время и посмотрим, насколько быстро можно создать простой блог, используя Django. (Стремление к совершенству постараемся учесть позже.)

Примечание

В этой главе предполагается, что вы уже установили платформу Django в своей системе. Если вы этого еще не сделали, обращайтесь к приложению В «Установка и запуск Django».

Вся работа в этой главе будет выполняться в командной строке (bash, tcsh, zsh, Cygwin или другой, по вашему выбору). Поэтому откройте терминал и перейдите в каталог, который включен в переменную окружения PYTHONPATH. В UNIX-подобных системах, таких как Linux, Mac OS X, FreeBSD и других, можно выполнить команду echo $PYTHONPATH, чтобы увидеть ее содержимое. В Win32 в окне приложения Командная строка (Command Prompt) выполните команду echo %PYTHONPATH%. Дополнительно о путях можно прочитать в главе о Python и в приложении, где описывается установка.

Мы рекомендуем вам попытаться в процессе чтения главы действительно создать блог. Если это практически невозможно – под рукой нет компьютера или вам не хватает терпения, простое чтение этой главы также будет полезным. Это особенно верно, если вы уже обладаете опытом работы с другими современными веб-платформами, поскольку основные понятия вам уже знакомы.

Если вы решите *создать* приложение на своем компьютере и достигнете момента, когда получаемые вами результаты будут расходиться с тем, что показано здесь, остановитесь и еще раз проверьте последний сделанный шаг, а затем загляните на два-три шага назад. Ищите недопонятую инструкцию или место, которое вы могли проскочить, потому что посчитали его малозначительным. Если озарение не придет, удалите проект и начните с самого начала. Авторы использовали этот метод при изучении платформы Django; мало того, что оказывается эффективнее начать все сначала, чем покорно и долго созерцать сообщения об ошибках, – повторение этапов, пройденных до момента, где пришлось столкнуться с ошибкой, действительно помогает отыскать ее!

Создание проекта

Простейший способ организовать программный код приложения на платформе Django состоит в том, чтобы использовать то, что в Django называется проектом: каталог файлов, которые обычно составляют отдельный веб-сайт. В состав Django входит утилита `django-admin.py`, которая упрощает решение таких задач, как создание дерева каталогов проекта. В операционной системе UNIX эта утилита по умолчанию устанавливается в каталог `/usr/bin`, а если вы пользуетесь системой Win32, она находится в папке `Scripts`, в каталоге установки Python – например, `C:\Python25\Scripts`. В любом случае вам необходимо убедиться, что утилита `django-admin.py` находится в одном из каталогов, включенном в переменную окружения `PATH`, чтобы ее можно было вызвать из командной строки.

Чтобы создать каталог проекта блога, выполните следующую команду:

```
django-admin.py startproject mysite
```

В Win32 вам необходимо открыть окно программы Командная строка (Command Prompt). Запустить эту программу можно, выбрав пункт меню Пуск→Все программы→Стандартные→Командная строка (Start→Programs→Accessories→Command Prompt). Вместо строки приглашения к вводу `$` вы увидите нечто следующее: `C:\WINDOWS\system32>`.

Теперь взгляните на содержимое каталога, который только что был создан этой командой. В системе UNIX оно должно выглядеть, как показано ниже:

```
$ cd mysite
$ ls -l
total 24
-rw-r--r--    1 pbx  pbx     0 Jun 26 18:51 __init__.py
-rwxr-xr-x    1 pbx  pbx   546 Jun 26 18:51 manage.py
-rw-r--r--    1 pbx  pbx  2925 Jun 26 18:51 settings.py
-rw-r--r--    1 pbx  pbx   227 Jun 26 18:51 urls.py
```

Если вы занимаетесь разработкой на платформе Win32, откройте окно приложения Проводник (Explorer). Содержимое каталога в окне этого приложения должно выглядеть, как показано на рис. 2.1, – при условии, что для размещения проекта вами была создана папка `C:\py\django`.

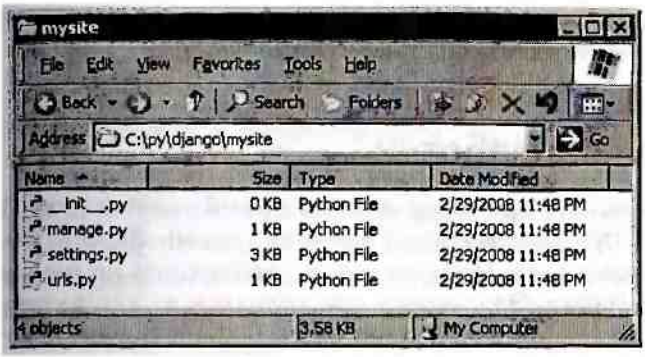

Рис. 2.1. Папка mysite в Win32

Примечание

Если вы обладаете опытом работы с языком Python, то вероятно уже знаете, что файл `__init__.py` превращает каталог проекта в **пакет** Python – набор модулей на языке Python. Тот факт, что это пакет, дает нам возможность использовать точечную нотацию для обращения к отдельным частям проекта, например `mysite.urls`. (Дополнительно о пакетах можно прочитать в главе 1, «Практическое введение в Python для Django».)

Помимо файла `__init__.py` команда `startproject` создала еще три файла.

- `manage.py` – утилита управления проектом Django. Взглянув на ее флаги прав доступа, можно увидеть, что она является выполняемой программой. Мы запустим ее чуть ниже.
- `settings.py` – файл, содержащий параметры настройки проекта со значениями по умолчанию. Здесь находится информация о базе данных, флаги отладки и другие важные переменные. Любое значение в этом файле доступно для всех приложений в проекте – мы увидите всю значимость этого в процессе чтения главы.
- `urls.py` – это то, что в платформе Django известно как URLconf, конфигурационный файл, который отображает шаблоны адресов URL на действия, выполняемые приложением. Этот файл является одной из самых замечательных и мощных особенностей Django.

Примечание

Любой файл, созданный командой `startproject`, содержит исходный программный код на языке Python. Среди них нет ни файлов в формате XML, ни

файлов .ini, и в них не используется замысловатый синтаксис описания конфигурации. Платформа Django стремится следовать философии применения «чистого языка Python» везде, где только возможно. Это обеспечивает высокую гибкость без добавления к платформе сложности. Например, если окажется необходимо, чтобы файл с параметрами настройки импортировал какой-нибудь *другой* файл или вычислял значения параметров вместо использования жестко установленных значений, вам ничто не помешает сделать это, так как это всего лишь вопрос программирования на языке Python.

Запуск сервера разработки

К настоящему моменту приложение блога еще не создано, и, тем не менее, платформа Django предлагает несколько компонентов, которые уже можно использовать. Один из таких удобнейших компонентов – встроенный веб-сервер. Этот сервер предназначен не для развертывания полноценных сайтов, а для разработки. В число преимуществ, получаемых от его использования, входят:

- Отсутствие необходимости устанавливать веб-сервер Apache, Lighttp или какой-либо другой веб-сервер, который мог бы использоваться для развертывания полноценных сайтов. Это особенно ценно, когда вы ведете работу на новом сервере, или занимаетесь разработкой на машине, которая не является сервером, или просто экспериментируете.
- Он автоматически определяет изменение исходных файлов с программным кодом на языке Python и перезагружает измененные модули. Это позволяет сэкономить *массу* времени на перезапуске веб-сервера вручную всякий раз, когда приходится править программный код, что является необходимым при использовании большинства других веб-серверов, использующих сценарии на языке Python.
- Он знает, как отыскивать и отображать графические файлы для приложения администрирования, что позволяет сразу же включить их в работу.

Запуск сервера разработки выполняется очень просто, всего одной командой. В процессе работы над нашим проектом мы будем использовать утилиту manage.py, сценарий-обертку, что избавит нас от необходимости указывать утилите django-admin.py, какой файл с настройками проекта следует использовать. Команда запуска сервера разработки выглядит, как показано ниже:

```
./manage.py runserver      # или ".\manage.py runserver" в win32
```

Вы должны увидеть примерно такой вывод, как показано ниже, с некоторыми отличиями при использовании платформ Win32, где для прерывания работы программы используется комбинация клавиш CTRL-BREAK, а не CONTROL-C:

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
(Перевод:
Проверка моделей...
0 ошибок найдено

Django версия 1.0, используются настройки 'mysite.settings'
Сервер разработки запущен для адреса http://127.0.0.1:8000/
Завершить работу сервера можно комбинацией клавиш CONTROL-C.)
```

Введя указанный адрес в своем броузере, вы должны увидеть страницу «It Worked!» (сервер работает), как показано на рис. 2.2.

Если при этом посмотреть на окно терминала, можно увидеть сообщение о том, что сервер зарегистрировал ваш запрос GET.

```
[07/Dec/2007 10:26:37] "GET / HTTP/1.1" 404 2049
```

Эта строка состоит из следующих четырех полей, слева направо: время, запрос, код HTTP ответа и количество байтов. (У вас количество байтов, скорее всего, будет немного отличаться от указанного здесь.) В данном случае был получен код ответа – 404 («Not Found» – страница не найдена), потому что в проекте пока еще не определен ни один адрес URL. Страница «It Worked!» – это просто доброжелательный способ сообщить вам об этом.

Совет

Если ваш сервер не заработал, вернитесь назад. Будьте безжалостны! Гораздо проще полностью удалить проект и начать чтение главы с самого начала, чем кропотливо проверять каждый файл и каждую строку программного кода.

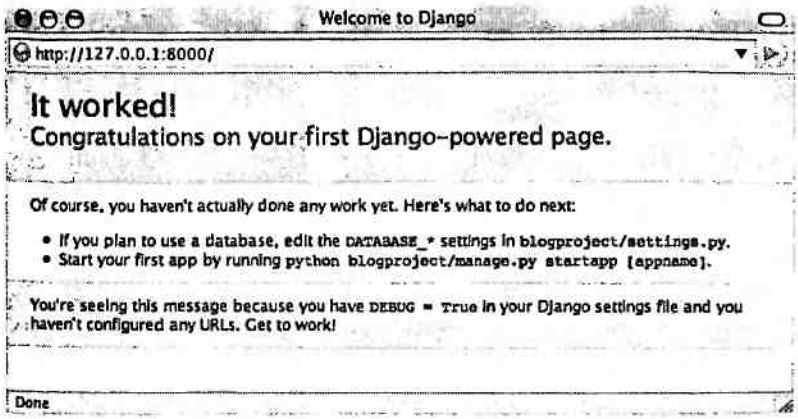

Рис. 2.2. Начальная страница «It Worked!» платформы Django

После благополучного запуска сервера можно переходить к разработке вашего первого приложения на платформе Django.

Создание приложения блога

Теперь, когда у нас имеется проект, можно приступать к созданию приложений в его рамках. Чтобы создать приложение блога, мы опять воспользуемся утилитой manage.py.

```
./manage.py startapp blog     # или ".\manage.py startapp blog" в win32
```

Это так же просто, как создание проекта. Теперь внутри каталога проекта у нас появился каталог blog. Ниже показано, что находится в этом каталоге, – сначала в формате листинга, полученного в системе UNIX, а затем на снимке с экрана (рис. 2.3), полученном в операционной системе Windows.

```
$ ls -l blog/
total 16
-rw-r--r--    1 pbx  pbx    0 Jun 26 20:33 __init__.py
-rw-r--r--    1 pbx  pbx   57 Jun 26 20:33 models.py
-rw-r--r--    1 pbx  pbx   26 Jun 26 20:33 views.py
```

Как и проект, приложение также является пакетом. Файлы models.py и views.py пока не содержат действующего программного кода, – это просто заготовки. При создании нашего простого блога нам не потребуется что-то менять в пустом файле views.py.

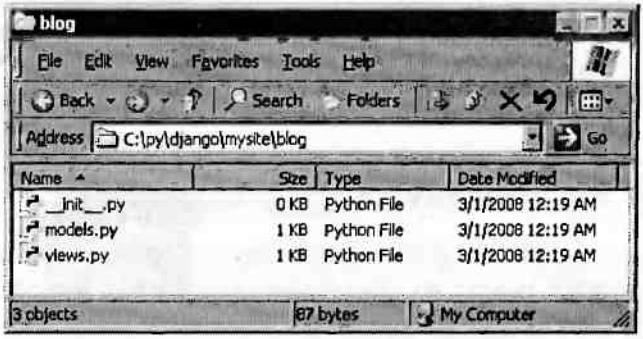

Рис. 2.3. Папка mysite\blog в Win32

Чтобы указать платформе Django, что это новое приложение является частью вашего проекта, вам необходимо отредактировать файл settings.py (который можно называть «файлом с настройками»). Откройте файл с настройками в своем текстовом редакторе и отыщите ближе к концу кортеж INSTALLED_APPS. Добавьте в этот кортеж имя приложения с использованием точечной нотации, как показано ниже (обратите внимание на завершающий символ запятой):

```
'mysite.blog',
```

С помощью кортежа INSTALLED_APPS платформа Django определяет настройки различных частей системы, включая приложение администрирования и платформу тестирования.

Создание модели

Теперь вы достигли ядра вашего приложения блога на платформе Django: файла models.py. Здесь определяются структуры данных блога. Следуя принципу DRY (Don't Repeat Yourself – не повторяйтесь), платформа Django получает массу сведений из описания модели вашего приложения. Давайте создадим простую модель, а затем посмотрим, что делает платформа Django на основе этой информации.

Откройте файл models.py в своем текстовом редакторе (предпочтительнее использовать редактор, который обладает функцией подсветки синтаксиса и поддерживает редактирование программного кода на языке Python). Вы должны увидеть следующий текст:

```
from django.db import models

# Create your models here.
(# Создайте свои модели здесь.)
```

Удалите комментарий и добавьте следующие строки:

```
class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()
```

Это законченная модель, представляющая объект «BlogPost» с тремя полями. (Строго говоря, этот объект содержит четыре поля – по умолчанию платформа Django автоматически создает автоинкрементное уникальное поле id для каждой модели.)

Как видите, наш вновь созданный класс BlogPost является подклассом django.db.models.Model. Это стандартный базовый класс платформы Django для моделей данных. Он является ядром мощной системы объектно-реляционного отображения. Кроме того, обратите внимание, что поля определяются как обычные атрибуты класса, каждый из которых является экземпляром определенного класса поля. Эти классы полей определяются в модуле django.db.models, где помимо трех типов, использованных здесь, находятся определения множества других типов, начиная от BooleanField и заканчивая XMLField.

Настройка базы данных

Если у вас нет настроенного и работающего сервера баз данных, мы рекомендуем задействовать SQLite, как наиболее простой и быстрый

способ обеспечить работу приложения. Это быстрая, легко доступная база данных, хранящая все данные в единственном файле. Управление доступом осуществляется на основе прав доступа к файлу. За дополнительной информацией о том, как настроить базу данных для использования в платформе Django, обращайтесь к приложению B.

Если у вас уже *имеется* сервер базы данных – PostgreSQL, MySQL, Oracle, MSSQL и вы хотите использовать его вместо SQLite, то с помощью своих инструментов администрирования создайте новую базу данных для своего проекта Django. В нашем примере мы присвоили своей базе данных имя «djangodb», но вы можете выбрать любое другое имя.

В любом случае, все, что останется сделать после создания пустой базы данных, – это сообщить платформе Django, как ее использовать. В этом месте на сцену выходит файл `settings.py`.

Использование сервера баз данных

Многие используют платформу Django совместно с серверами реляционных баз данных, такими как PostgreSQL или MySQL. В файле с настройками существует шесть параметров, имеющих отношение к базе данных (хотя вы, скорее всего, будете использовать только два): `DATABASE_ENGINE`, `DATABASE_NAME`, `DATABASE_HOST`, `DATABASE_PORT`, `DATABASE_USER` и `DATABASE_PASSWORD`. Имена параметров говорят сами за себя. От вас требуется лишь определить корректные значения, соответствующие серверу баз данных, используемому для нужд платформы Django. Например, ниже приводятся настройки базы данных MySQL:

```
DATABASE_ENGINE = "mysql"
DATABASE_NAME = "djangodb"
DATABASE_HOST = "localhost"
DATABASE_USER = "paul"
DATABASE_PASSWORD = "pony"  # секрет!
```

Примечание

Мы не определили значение параметра `DATABASE_PORT`, потому что это необходимо, только если сервер баз данных использует нестандартный порт. Например, сервер MySQL по умолчанию использует порт 3306. Если вы не изменяли эту настройку сервера, то можно вообще не определять параметр `DATABASE_PORT`.

За дополнительной информацией о создании новой базы данных и нового пользователя (необходимо для серверов баз данных) обращайтесь к приложению B.

Использование SQLite

SQLite – это популярный механизм баз данных, часто используемый для нужд тестирования и даже для работы, когда не приходится иметь дело с большим числом одновременных попыток записи данных. При

использовании этого механизма не требуется указывать имя хоста, номер порта, имя пользователя и пароль, потому что в качестве хранилища информации SQLite использует локальную файловую систему, а управление доступом к данным осуществляется на основе системы прав доступа к файлу. Поэтому при использовании базы данных SQLite достаточно сообщить платформе Django всего два параметра.

```
DATABASE_ENGINE = "sqlite3"
DATABASE_NAME = "/var/db/django.db"
```

Примечание

При использовании SQLite совместно с настоящим веб-сервером, таким как Apache, необходимо убедиться, что учетная запись, под которой выполняется процесс веб-сервера, обладает правом записи как в файл базы данных, так и в *каталог, где находится этот файл*. Для сервера разработки, который мы используем в данном случае, это не является проблемой, потому что пользователь (то есть вы), запустивший сервер, также является владельцем файлов и каталогов проекта.

SQLite также является популярным выбором для платформы Win32, потому что этот механизм входит в состав дистрибутива Python. С учетом того, что мы уже создали каталог проекта (и приложения) C:\py\django, создадим также каталог db.

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = r'C:\py\django\db\django.db'
```

Если вы еще плохо знакомы с языком Python, обратите внимание на малозаметное отличие между двумя последними примерами – в первом примере мы заключили имя sqlite3 в кавычки, тогда как в примере для Win32 мы использовали апострофы. Это не имеет никакого отношения к различиям между платформами – в языке Python отсутствует тип данных «символ», поэтому апострофы и кавычки интерпретируются совершенно одинаково. От вас требуется лишь, чтобы открывающая и закрывающая кавычка были одного и того же типа!

Вы должны были также заметить маленький символ «r» перед именем папки. Если вы читали главу 1, то должны помнить, что этот символ является спецификатором объекта «сырой строки», то есть такой строки, в которой все символы интерпретируются буквально, без преобразования специальных символов. Например, комбинация символов \n обычно используется для обозначения символа перевода строки, но в «сырых» строках она интерпретируется (буквально) как два отдельных символа: символ обратного слеша, за которым следует символ n. Таким образом, назначение «сырой» строки, используемой, в частности, для определения путей к файлам в DOS, состоит в том, чтобы предотвратить преобразование специальных символов (если таковые имеются в строке) интерпретатором Python.

Создание таблиц

Теперь можно предложить платформе Django задействовать информацию, представленную вами, чтобы подключиться к базе данных и создать необходимые таблицы. Делается это с помощью простой команды:

```
./manage.py syncdb    # или ".\manage.py syncdb" в win32
```

По мере того как платформа Django будет подготавливать базу данных, вы увидите вывод, который начинается примерно так, как показано ниже:

```
                                    Перевод:
Creating table auth_message         Создается таблица auth_message
Creating table auth_group           Создается таблица auth_group
Creating table auth_user            Создается таблица auth_user
Creating table auth_permission      Создается таблица auth_permission
Creating table django_content_type  Создается таблица django_content_type
Creating table django_session       Создается таблица django_session
Creating table django_site          Создается таблица django_site
Creating table blog_blogpost        Создается таблица blog_blogpost)
```

После запуска команды syncdb платформа Django просматривает содержимое файла models.py, отыскивая все установленные приложения, перечисленные в параметре INSTALLED_APPS. Для каждой найденной модели она создает в базе данных отдельную таблицу. (Из этого правила есть исключения, которые мы рассмотрим ниже, когда будем обсуждать такие причудливые особенности, как организация отношений «многие ко многим», но для нашего случая это правило действует. Если вы используете SQLite, вы также должны заметить, что файл django.db базы данных был создан именно там, где вы указали.)

Для всех элементов в параметре INSTALLED_APPS, которые присутствовали там по умолчанию, также имеются свои модели. Вывод команды manage.py syncdb подтверждает, что платформа Django создала по одной или более таблиц для каждого приложения.

Однако мы привели не весь вывод команды syncdb. Кроме того, вам будет задано несколько вопросов, имеющих отношение к приложению django.contrib.auth.

```
You just installed Django's auth system, which means you don't have
any superusers defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'pbx'):
E-mail address: pb@e-scribe.com
Password:
Password (again):
Superuser created successfully.
Installing index for auth.Message model
Installing index for auth.Permission model
(Перевод:
Вы только что установили систему аутентификации Django; это означает,
что у вас пока нет учетных записей суперпользователей.
```

```
Хотите создать такую учетную запись сейчас? (да/нет): да
Имя пользователя (Оставьте строку пустой, чтобы использовать имя 'pbx'):
Адрес электронной почты: pb@e-scribe.com
Пароль:
Пароль (еще раз):
Учетная запись суперпользователя была создана.
Устанавливается индекс для модели auth.Message
Устанавливается индекс для модели auth.Permission)
```

Теперь в системе аутентификации у вас имеется учетная запись суперпользователя (надеемся, что это вы сами). Она понадобится нам через минуту, когда мы будем добавлять автоматизированное приложение администрирования.

Этот процесс завершается парой строк, имеющих отношение к особенности, называемой **оснастка (fixtures)**, к которой мы вернемся в главе 4, «Определение и использование моделей». Эта особенность обеспечивает возможность предварительной загрузки данных во вновь созданном приложении. Сейчас мы не задействуем эту особенность, поэтому платформа просто движется дальше.

```
Loading 'initial_data' fixtures...
No fixtures found.
(Перевод:
Загружается оснастка 'initial_data'
Оснастка не найдена.)
```

На этом первичная подготовка базы данных закончена. Когда в следующий раз вы будете запускать команду syncdb для данного проекта (что придется делать всякий раз после добавления нового приложения или модели), объем вывода будет немного меньше, потому что уже созданные таблицы пересоздаваться не будут и также не будет выводиться запрос с предложением создать учетную запись суперпользователя.

Настройка автоматизированного приложения администрирования

Автоматизированное приложение администрирования часто называют «драгоценным камнем в короне» Django. Для тех, кто уже устал создавать интерфейсы CRUD (Create, Read, Update, Delete – создать, прочитать, обновить, удалить) для веб-приложений, это приложение – просто подарок. Поближе с этим приложением мы познакомимся в разделе «Настройка приложения администрирования» в главе 11, «Передовые приемы программирования в Django». А сейчас мы просто активируем его и опробуем.

Так как это приложение не относится к разряду обязательных, его требуется указать в файле settings.py, точно так же, как вы делали это со своим собственным приложением блога. Откройте файл settings.py и добавьте следующую строку в кортеж INSTALLED_APPS, сразу вслед за 'django.contrib.auth':

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

Всякий раз после добавления нового приложения в свой проект, вы должны запустить команду syncdb, чтобы обеспечить создание всех необходимых таблиц в вашей базе данных. Сейчас мы добавили приложение администрирования в кортеж INSTALLED_APPS, и поэтому вызов команды syncdb приводит к созданию еще одной таблицы в базе данных:

```
$ ./manage.py syncdb
Creating table django_admin_log
Installing index for admin.LogEntry model
Loading 'initial_data' fixtures...
No fixtures found.
```

Теперь, когда приложение настроено, нам необходимо определить адрес URL, чтобы можно было получить доступ к этому приложению. Обратите внимание на следующие строки в автоматически сгенерированном файле urls.py. Вам наверняка потребуется раскомментировать строки 2, 3 и 16 (и, вероятно, 12).

```
# Раскомментируйте следующие две строки для раздела администратора:
from django.contrib import admin
admin.autodiscover()

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

    # Раскомментируйте строку admin/doc ниже и добавьте
    # 'django.contrib.admindocs' в кортеж INSTALLED_APPS,
    # чтобы разрешить доступ к документации администратора:
    (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Раскомментируйте следующую строку, чтобы активировать
    # административный раздел:
    (r'^admin/(.*)', admin.site.root),
)
```

Удалите символ # во второй строке (одновременно вы можете полностью удалить первую строку, которая является просто комментарием) и сохраните файл. Этим вы сообщаете платформе Django, что она должна загрузить административный раздел сайта, являющийся специальным объектом, который используется вспомогательным приложением администрирования.

Наконец, ваши приложения должны сообщить платформе, какие модели должны быть доступны для редактирования на страницах административного раздела. Для этого достаточно просто определить административный раздел сайта по умолчанию, упомянутый выше, и зарегистрировать в нем модель BlogPost. Откройте файл mysite/blog/models.py, добавьте инструкцию импортирования приложения администрирования и затем добавьте в конец файла строку, выполняющую регистрацию вашей модели.

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

admin.site.register(BlogPost)
```

Этот пример простейшего использования приложения администрирования представляет собой лишь вершину айсберга – создав специальный класс `Admin` для заданной модели, можно определить массу различных параметров, имеющих отношение к выполнению административных задач, и затем зарегистрировать модель с этим классом. Вскоре мы воспользуемся такой возможностью, а в последующих главах вы увидите примеры более сложного использования приложения администрирования, особенно в части III «Приложения Django в примерах» и IV «Дополнительные возможности и особенности Django».

Опробование приложения администрирования

Теперь, когда мы оснастили свой сайт Django приложением администрирования и зарегистрировали в нем нашу модель, можно попробовать покрутиться в нем. Выполните команду `manage.py runserver` еще раз. Теперь откройте в своем веб-броузере страницу *http://127.0.0.1:8000/admin/*. (Не волнуйтесь, если у вас адрес сервера разработки отличается от того, что указан здесь, просто добавьте к нему строку *admin/*.) Вы должны увидеть страницу регистрации, как показано на рис. 2.4.

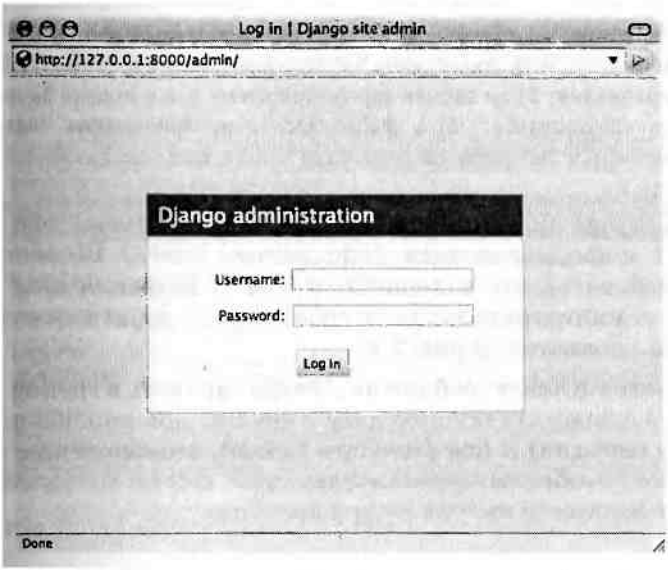

Рис. 2.4. Страница регистрации приложения администрирования

Введите в поле Username (имя пользователя) имя «суперпользователя» и пароль в поле Password (пароль), созданные ранее. После регистрации вы должны увидеть домашнюю страницу администратора, как показано на рис. 2.5.

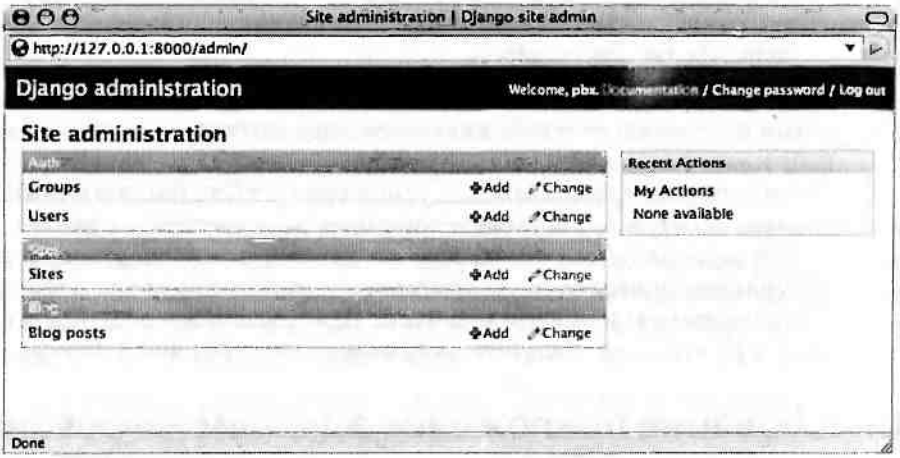

Рис. 2.5. Домашняя страница администратора

Мы выполним экскурс по этому интерфейсу далее в книге, а пока просто убедитесь, что ваше приложение Blog присутствует на странице, как показано на снимке с экрана. Если его там нет, проверьте, правильно ли вы выполнили предыдущие шаги.

Совет

Наиболее часто отсутствие приложения на странице администратора обусловлено тремя причинами: 1) вы забыли зарегистрировать класс модели вызовом метода `admin.site.register`, 2) в файле `models.py` приложения имеются ошибки, 3) вы забыли добавить приложение в кортеж `INSTALLED_APPS`, в свой файл `settings.py`.

А как быть с информационным наполнением блога? Щелкните на кнопке Add (добавить), что находится справа от названия Blog Posts. Приложение администрирования откроет форму добавления нового сообщения, как показано на рис. 2.6.

Введите заголовок и текст сообщения. Чтобы записать в группу полей Timestamp (дата и время) текущую дату и время, можно щелкнуть на ссылках Today (сегодня) и Now (текущее время). Можно также щелкнуть на ярлыке с изображением календаря или часов и воспользоваться удобными средствами выбора даты и времени.

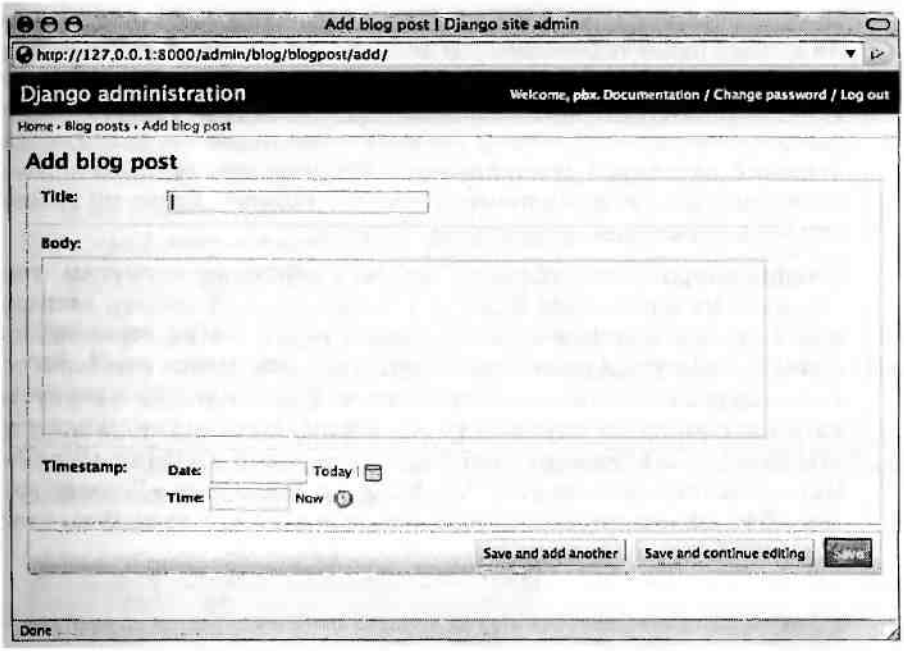

Рис. 2.6. Добавление информационного наполнения из приложения администрирования

После того как вы сочините текст своего шедевра, щелкните на кнопке Save (сохранить). После этого вы увидите страницу с текстом подтверждения («The blog post 'BlogPost object'was added successfully» – 'объект BlogPost' с сообщением был успешно добавлен) и список всех сообщений, общее число которых равно 1, как показано на рис. 2.7.

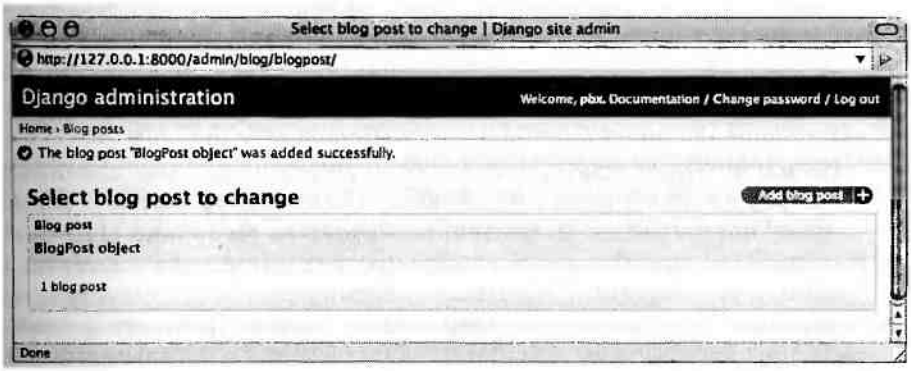

Рис. 2.7. Первое сообщение в блоге было успешно сохранено

Почему сообщение названо так неуклюже 'BlogPost object'? Платформа Django проектировалась специально, чтобы обеспечить поддержку бесконечного разнообразия типов информационного наполнения, поэтому она не делает никаких предположений о том, какое поле лучше всего соответствует данному элементу содержимого. В примерах приложений из третьей части книги вы увидите, как указывать желаемое поле или даже конструировать строку, которая будет по умолчанию использоваться для обозначения объекта.

Теперь попробуйте добавить второе сообщение с другим текстом, щелкнув на кнопке Add Blog Post + (добавить сообщение), расположенной в правом верхнем углу. Когда вы вернетесь на страницу со списком, вы сразу заметите, что на странице появилась еще одна строка. Если попробовать обновить страницу или выйти из нее и вернуться обратно, это никак не улучшит то, что вы видите, – вы не сможете почувствовать удовлетворение, наблюдая на экране заголовки «BlogPost object», как показано на рис. 2.8. Если вы подумали: «Должен же быть способ сделать отображение более наглядным!», – то знайте, что вы не первый об этом подумали.

Рис. 2.8. Не слишком полезная итоговая страница

Но мы не будем ждать, пока отображение списка само изменится. Ранее мы активировали приложение администрирования с минимальными настройками, а именно, регистрировали только одну нашу модель. Однако, добавив две строки программного кода и изменив вызов метода регистрации, мы можем обеспечить более наглядное и более полезное представление элементов в списке. Добавьте в свой файл `mysite/blog/models.py` новый класс `BlogPostAdmin` и добавьте его имя в вызов метода регистрации так, чтобы теперь содержимое файла `models.py` выглядело, как показано ниже:

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

class BlogPostAdmin(admin.ModelAdmin):
    list_display = ('title', 'timestamp')

admin.site.register(BlogPost, BlogPostAdmin)
```

Сервер разработки обнаружит изменения и перезагрузит файл моделей. Если в этот момент вы посмотрите в окно терминала, то увидите соответствующие сообщения об этом.

Если теперь обновить страницу, можно будет увидеть более информативное содержимое, основанное на переменной `list_display`, добавленной в класс `BlogPostAdmin` (рис. 2.9).

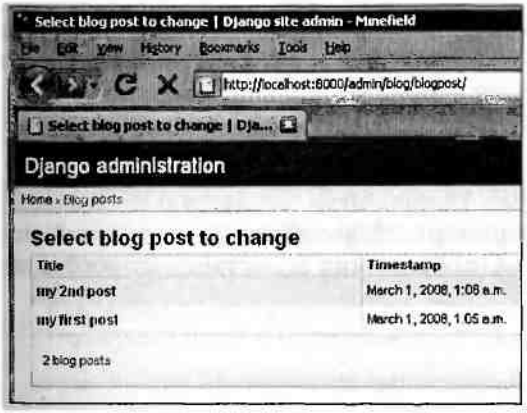

Рис. 2.9. Так намного лучше

Попробуйте щелкнуть на заголовках столбцов Title и Timestamp – каждый из них отвечает за сортировку элементов списка. Например, щелчок на заголовке Title приведет к выполнению сортировки по заголовкам сообщений в порядке возрастания, повторный щелчок на заголовке Title приведет к выполнению сортировке в порядке *убывания*.

В приложении администрирования имеется много других полезных особенностей, которые можно активировать одной или двумя строками программного кода: поиск, сортировка по собственным критериям, фильтрация и многое другое. Как уже неоднократно упоминалось, в третьей и четвертой частях книги многие из этих тем будут рассматриваться более подробно.

Создание общедоступного раздела приложения блога

После создания базы данных и административного раздела нашего приложения пришло время заняться общедоступными страницами. С точки зрения платформы Django, страница состоит из трех компонентов:

- Шаблон, отображающий информацию, переданную ему (в виде объекта словаря с именем `Context`)
- Функция представления, которая выбирает информацию для отображения, как правило, из базы данных
- Шаблон адреса URL, с помощью которого определяется соответствие между входящим запросом и функцией представления, а также с возможными дополнительными параметрами для функции представления

Мы рассмотрим все три компонента в указанном порядке. В некотором смысле этот порядок соответствует направлению изнутри наружу – когда платформа Django обрабатывает запрос, она начинает с шаблонов адресов URL, затем вызывает функцию представления и возвращает данные, отображенные в шаблон.

Создание шаблона

Язык шаблонов платформы Django читается настолько легко, что мы можем сразу перейти к примеру. Ниже приводится простой шаблон, который предназначен для отображения единственного сообщения:

```
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
```

Это обычный код разметки HTML (хотя шаблоны Django могут использоваться для вывода текстовой информации в любом формате), в который добавлены специальные теги шаблона, заключенные в фигурные скобки. Это **теги переменных**, которые отображают данные, переданные шаблону. Внутри таких тегов можно использовать точечную нотацию, принятую в языке Python, для доступа к атрибутам объектов, передаваемых шаблону. Например, в данном шаблоне предполагается, что ему передали объект типа `BlogPost` с именем `post`. Три строки шаблона извлекают из объекта `BlogPost` поля `title`, `timestamp` и `body` соответственно.

Давайте немного расширим шаблон так, чтобы он мог использоваться для отображения множества сообщений, для чего воспользуемся тегом `for` шаблонов Django.

```
{% for post in posts %}
<h2>{{ post.title }}</h2>
```

```
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
{% endfor %}
```

Первые три строки остались без изменений – мы просто добавили **тег блока** с именем for, который будет выполнять заключенный в него раздел шаблона для каждого элемента последовательности. Синтаксис тега специально был выбран так, чтобы сделать его похожим на синтаксис цикла for языка Python. Обратите внимание, что в отличие от тегов переменных теги блоков заключаются в пару последовательностей символов {% ... %}.

Сохраните этот простой шаблон, состоящий из пяти строк, в файле с именем archive.html в каталоге templates, внутри каталога blog вашего приложения. То есть путь к шаблону должен выглядеть так:

```
mysite/blog/templates/archive.html
```

Имя самого шаблона можно выбирать совершенно произвольно (мы с таким же успехом могли назвать его foo.html), но имя каталога обязательно должно быть templates. Когда платформа Django пытается отыскать шаблон, она по умолчанию просматривает каталоги templates всех установленных приложений.

Создание функции представления

Теперь напишем простую функцию представления, которая будет извлекать все сообщения из базы данных и отображать их с помощью нашего шаблона. Откройте файл blog/views.py и добавьте в него следующие строки:

```
from django.template import loader, Context
from django.http import HttpResponse
from mysite.blog.models import BlogPost

def archive(request):
    posts = BlogPost.objects.all()
    t = loader.get_template("archive.html")
    c = Context({ 'posts': posts })
    return HttpResponse(t.render(c))
```

Пропустим строки с инструкциями import (они лишь загружают необходимые нам функции и классы) и рассмотрим остальные строки более подробно:

- Строка 5: Любая функция представления в платформе Django принимает в виде первого аргумента объект типа django.http.HttpRequest. Она также может принимать и другие аргументы, которые передаются посредством шаблона адреса URL, причем такая возможность будет использоваться вами очень часто.
- Строка 6: Класс BlogPost, созданный нами как подкласс django.db.models.Model, унаследовал всю мощь системы объектно-реля-

ционного отображения. Эта строка представляет собой простой пример использования **ORM** (Object-Relational Mapper – объектно-реляционное отображение; подробнее об этом рассказывается в главе 3 «Начало» и в главе 4) для извлечения из базы данных всех объектов `BlogPost`.

- Строка 7: Чтобы создать объект шаблона `t`, достаточно просто сообщить платформе Django имя шаблона. Поскольку мы сохранили шаблон в каталоге `templates`, Django отыщет его без дополнительной помощи с нашей стороны.
- Строка 8: Шаблоны Django отображают данные, которые передаются им в виде объекта словаря. Наш объект `c` имеет всего один ключ и значение.
- Строка 9: Любая функция представления должна возвращать объект типа `django.http.HttpResponse`. В простейшем случае конструктору передается обычная строка, а метод `render` шаблона возвращает необходимую строку.

Создание шаблона адреса URL

Для нормальной работы нашей странице, как и всему остальному в Интернете, необходим адрес URL.

Мы могли бы создать требуемый шаблон URL прямо в файле `mysite/urls.py`, но это создаст ненужную связь между проектом и приложением. Приложение блога может использоваться нами еще где-нибудь, поэтому лучше будет сделать так, чтобы все определения URL приложения находились в самом приложении. Для этого мы сделаем два простых шага.

Первый шаг напоминает порядок активизации приложения администрирования. В файле `mysite/urls.py` присутствует закомментированная строка с примером, которая содержит практически все, что нам потребуется. Отредактируйте ее так, как показано ниже:

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

Она будет перехватывать все запросы, начинающиеся с *blog/*, и передавать их новому модулю настройки шаблонов URL, который мы создадим ниже.

Второй шаг заключается в определении адресов URL внутри пакета приложения блога. Создайте новый файл `mysite/blog/urls.py` и добавьте в него следующие строки:

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

urlpatterns = patterns('',
    url(r'^$', archive),
)
```

Он выглядит очень похожим на основной файл шаблонов URL. Основное действие выполняется в строке 5. Но сначала обратите внимание на отсутствие части адреса *blog/*, соответствие с которой определялось в корневом файле шаблонов URL, – она была удалена, поскольку наше приложение подразумевает возможность многократного использования и потому не должно зависеть от того, к какому адресу верхнего уровня оно подключено, будь то *blog/*, *news/* или *what/i/had/for/lunch/*. Регулярное выражение в строке 5 соответствует пустому адресу URL, такому как */blog/*.

Функция представления `archive` передается во втором элементе кортежа шаблона. (Обратите внимание, что здесь передается не строка с именем функции, а сам объект функции. Однако строки с именами также могут использоваться, как будет показано ниже.)

Давайте теперь посмотрим, как действует наше приложение! Сервер разработки еще работает? Если нет, запустите его командой `manage.py runserver` и затем откройте в броузере страницу с адресом *http://127.0.0.1:8000/blog/*. Вы должны увидеть страницу со списком всех сообщений, введенных вами ранее, с заголовками, временем создания и телом сообщения.

Заключительные штрихи

Используя ключевые понятия, рассмотренные к настоящему моменту, вы сможете продвинуться дальше и внести в этот примитивный механизм блога некоторые усовершенствования. Давайте рассмотрим некоторые из них, чтобы придать нашему приложению немного лоска.

Усовершенствование шаблона

Наш шаблон настолько прост, что проще некуда. Поскольку эта книга рассматривает вопросы веб-программирования, а не веб-дизайна, оставим проблему эстетики на ваше усмотрение и поговорим о наследовании шаблонов – еще одной особенности системы шаблонов, которая может упростить вашу жизнь, особенно с увеличением числа стилей страниц.

Наш простой шаблон целиком и полностью является независимым. Но как быть, если на сайте имеется не только блог, но еще архив фотографий и страница со ссылками на другие ресурсы и нам необходимо, чтобы все страницы отображались в едином стиле? Опыт подсказывает вам *неправильный* путь, который заключается в том, чтобы создать три одинаковые, но независимые копии базового шаблона. *Правильный* способ, принятый в платформе Django, состоит в том, чтобы создать единый базовый шаблон, а затем *наследовать* его в других шаблонах. Создайте в каталоге `mysite/blog/templates` шаблон с именем `base.html` и добавьте в него следующие строки:

```
<html>
<style type="text/css">
body { color: #efd; background: #453; padding: 0 5em; margin: 0 }
h1 { padding: 2em 1em; background: #675 }
h2 { color: #bf8; border-top: 1px dotted #fff; margin-top: 2em }
p { margin: 1em 0 }
</style>
<body>
<h1>mysite.example.com</h1>
{% block content %}
{% endblock %}
</body>
</html>
```

Этот шаблон не совсем точно следует требованиям спецификации XHTML Strict, но он работает. Обратите внимание на тег `{% block ... %}`. Он определяет именованную область, которая может изменяться дочерними шаблонами. Чтобы задействовать этот шаблон в приложении блога, необходимо изменить шаблон `archive.html` так, чтобы он ссылался на этот новый базовый шаблон и его блок «content».

```
{% extends "base.html" %}
{% block content %}
{% for post in posts %}
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
{% endfor %}
{% endblock %}
```

Тег `{% extends ... %}` предписывает платформе Django отыскать шаблон с именем `base.html` и подставить содержимое всех именованных блоков *этого* шаблона на место соответствующих блоков *базового* шаблона. Теперь список сообщений должен выглядеть примерно так, как показано на рис. 2.10 (надеемся, что у вас сообщения содержат более интересную информацию).

Упорядочение по дате

Вы должны были заметить, что сообщения в блоге располагаются не в обратном хронологическом порядке, ставшем уже традиционным. Однако исправить это с помощью Django совсем несложно – нам только нужно выбрать, в каком месте это сделать. Мы можем добавить упорядочение по умолчанию в модель или определить порядок сортировки в функции представления – в вызове метода `BlogPost.objects.all()`. В данном случае модель является более предпочтительным местом, потому что чаще всего требуется, чтобы сообщения в блоге следовали в обратном хронологическом порядке. Если определить предпочтительный порядок следования внутри модели, все части Django, обращающиеся к нашим данным, будут использовать этот порядок.

Рис. 2.10. *Слегка стилизованный блог*

Чтобы определить порядок следования элементов в модели, необходимо добавить вложенный класс `Meta` и определить атрибут `ordering` в этом классе.

```
class Meta:
    ordering = ('-timestamp',)
```

Взгляните теперь на домашнюю страницу блога (*/blog/*). Самое последнее сообщение должно теперь находиться в начале списка. Строка `-timestamp` — это сокращенный способ сообщить платформе Django: «упорядочить по полю `'timestamp'` в порядке убывания». (Если опустить знак «-», сообщения будут отсортированы по времени в порядке возрастания.)

Примечание

Не опускайте завершающую запятую внутри круглых скобок! Благодаря ей значение справа от оператора присваивания интерпретируется как кортеж с единственным элементом, а не как простая строка. В данном случае платформа Django ожидает получить кортеж, в котором вы можете указать произвольное число полей, участвующих в сортировке. Если после запятой добавить `'title'`, и представить, что в блоге имеется два сообщения с заголовками «А» и «В» и с одинаковым временем создания, сообщение «А» будет располагаться в списке первым.

Форматирование даты и времени с помощью фильтра

Формат ISO8601 представления даты и времени не слишком привлекателен, поэтому попробуем выводить дату и время в более удобочитаемом виде, использовав для этого мощную особенность системы шаблонов: фильтры.

Поскольку эта особенность касается представления, а не структуры данных или бизнес-логики, наиболее подходящим местом для ее размещения является шаблон. Откройте файл archive.html и измените строку, содержащую атрибут post.timestamp, как показано ниже

```
<p>{{ post.timestamp|date }}</p>
```

Чтобы применить фильтр к переменной, нужно просто добавить его после имени переменной, отделив символом вертикальной черты. Перезагрузите домашнюю страницу блога. Теперь дата и время должны отображаться в более дружественном формате («July 7»).

Если стиль отображения, который по умолчанию дает фильтр date, вас не устраивает, можете передать ему дополнительный аргумент со строкой формата, как в функции strftime. Однако, вместо того чтобы задействовать коды преобразования из модуля time, фильтр date использует спецификаторы формата, применяемые в функции date языка PHP. Например, если необходимо отобразить день недели, но опустить год, измените строку, чтобы передать фильтру date аргумент, как показано ниже.

```
<p>{{ post.timestamp|date:"l, F jS" }}</p>
```

Этот конкретный пример строки форматирования позволяет выводить дату в виде «Friday, July 6th». Убедитесь, что с обеих сторон двоеточия нет пробелов – механизм шаблонов Django чувствителен к ним.

В заключение

Безусловно, мы могли бы бесконечно добавлять и добавлять новые особенности в наш блог (многие так и поступают!), но мы надеемся, что вы увидели уже достаточно, чтобы ощутить мощь платформы Django. В ходе создания этого простого приложения блога вы познакомились с некоторыми интересными особенностями Django, позволяющими сэкономить время:

- Встроенный веб-сервер, который обеспечивает независимость в процессе разработки и автоматически перезагружает ваш программный код после его правки
- Подход к созданию моделей основан исключительно на использовании языка Python, что избавляет вас от необходимости писать или сопровождать код SQL или файлы XML с описанием

- Автоматизированное приложение администрирования обеспечивает полноценную возможность редактирования информационного наполнения даже для неспециалистов
- Система шаблонов может использоваться для воспроизводства текста в форматах HTML, CSS, JavaScript и других
- Фильтры шаблонов позволяют видоизменять отображение данных (таких, как дата и время) без вмешательства в бизнес-логику приложения
- Система шаблонов URL обеспечивает высокую гибкость в разработке схемы адресов URL, сохраняя части адресов URL, имеющих отношение к приложению, которому они принадлежат

Только чтобы дать вам представление о том, что рассматривается далее, ниже перечислены дополнительные возможности, которые можно было бы добавить в наш блог, используя встроенные особенности Django:

- Публикация последних сообщений по каналам Atom и RSS (глава 11)
- Поиск, чтобы пользователи могли отыскивать сообщения, содержащие определенные фразы (смотрите пример приложения системы управления содержимым в главе 8 «Система управления содержимым»)
- Задействовать «универсальные функции представления» платформы Django, чтобы полностью избавиться от необходимости писать программный код в файле `views.py` (смотрите пример приложения Pastebin в главе 10 «Pastebin»)

Вы завершили краткий экскурс по основам платформы Django. В главе 3 более подробно и более широко рассматриваются ключевые компоненты платформы, а также философия, положенная в их основу. Кроме того, там дается краткий обзор некоторых принципов веб-разработки, занимающих важное место не только в самой платформе Django, но и в последующих главах книги. В главе 4 вы поближе познакомитесь с особенностями платформы и найдете ответы на вопросы «как, почему и что если...?», которые наверняка возникли у вас при изучении предыдущих примеров. После главы 4 у вас сложится четкое представление о платформе, достаточное, чтобы двигаться дальше и создать несколько примеров приложений: систему управления содержимым, pastebin, фотогалерею и «живой блог», использующий технологию Ajax.

3

Начало

Как и любые другие крупные программные проекты, платформа Django охватывает большое число концепций, особенностей и инструментов. Комплекс задач, для решения которых она создавалась, а именно задач разработки веб-приложений, также имеет немаловажное значение. Прежде чем приступить к детальному изучению платформы Django, вам необходимо понять стоящие перед вами задачи и методы, которые используют платформы, такие как Django, для их решения.

В этой главе будут представлены основные идеи, начиная с общего взгляда на Веб, без привязки к каким-либо инструментальным средствам, с последующим описанием модели веб-платформы и ее составных частей и объяснением общей философией разработки, используемой разработчиками Django. Некоторые представления, полученные вами в предыдущей главе, обретут для вас более конкретный смысл.

Важное примечание: если вы хорошо знаете основы разработки веб-приложений, некоторые из представленных здесь концепций будут вам хорошо знакомы, но даже опытные веб-разработчики смогут извлечь выгоду от повторения теоретических основ. Слишком часто наше мышление ограничено рамками применения определенного языка программирования или набора инструментальных средств, тогда как более широкая перспектива нередко открывает решения, ранее скрытые от взгляда.

Надежное овладение рассматриваемыми здесь базовыми концепциями поможет вам находить лучшие решения и расширит возможности выбора – как в процессе проектирования, так и в ходе реализации. Поэтому, пожалуйста, не пропускайте эту главу!

Основы динамических веб-сайтов

В своей основе разработка веб-приложений концептуально проста. Пользователи запрашивают у веб-сервера документ, веб-сервер извлекает или генерирует запрошенный документ, сервер возвращает результат броузеру пользователя, а броузер отображает его. В каждом конкретном случае детали могут изменяться, но общая схема остается неизменной. Попробуем разобраться с тем, как в эту схему встраиваются веб-платформы, такие как Django.

Взаимодействие: HTTP, URL, запросы, ответы

Протокол HTTP (HyperText Transfer Protocol – протокол передачи гипертекста) инкапсулирует процесс обслуживания веб-страниц и составляет основу Всемирной паутины. Поскольку этот протокол предназначен для организации взаимодействий между сервером и клиентом, он в значительной степени состоит из **запросов** (клиентов к серверу) и **ответов** (сервера клиентам). Все, что происходит на стороне сервера между запросом и ответом, уже не относится к протоколу HTTP и обеспечивается программным обеспечением сервера (смотрите ниже).

Понятие **запроса** составляет первый этап процесса – с его помощью клиент запрашивает у сервера некоторый документ. Основой запроса является адрес **URL** – «путь» к запрашиваемому документу, однако имеется несколько способов дополнять запросы параметрами, что обеспечивает возможность наделить один и тот же адрес URL различными вариантами поведения.

Ответ состоит, прежде всего, из **тела** – обычно это текст веб-страницы, и сопровождающих его **заголовков**, содержащих дополнительную информацию о данных – например, дата и время последнего изменения, как долго эта страница может сохраняться в локальном кэше, тип содержимого и т. д. Помимо разметки HTML ответ может содержать обычный текст, изображения, документы (PDF, Word, Excel и другие), аудиоданные и т. д.

Платформа Django представляет запросы и ответы в виде относительно простых объектов на языке Python с атрибутами для хранения различных элементов данных и с методами для выполнения более сложных операций.

Хранилища данных: SQL и реляционные базы данных

В упрощенном представлении Всемирная паутина занимается передачей данных, то есть обеспечивает совместное использование информации (в самом прямом смысле – записи в блогах, данные о финансовом положении, электронные книги и т. д.). На первых порах развития Всемирной паутины информационное наполнение состояло из текстовых файлов в формате HTML, написанных вручную и хранящихся

в файловых системах серверов. Такие страницы называются **статическими**, так как в ответ на запрос с одним и тем же адресом URL всегда возвращается одна и та же информация. «Пути», описанные выше, были более примитивными – они не имели параметров, поскольку они были просто путями к файлам в файловой системе сервера, где было расположено статическое информационное наполнение. В настоящее время большая часть содержимого носит **динамический** характер, потому что в ответ на запрос с одним и тем же адресом URL могут возвращаться совершенно разные данные, в зависимости от различных факторов.

В значительной степени динамическая природа обусловлена тем, что информация хранится в **базе данных**, которая обеспечивает возможность создавать элементы данных, состоящие из множества частей, и связывать их друг с другом, обозначая взаимоотношения между ними. Определение и запросы к базе данных осуществляются с помощью языка **SQL** (Structured Query Language – структурированный язык запросов), причем нередко с помощью уровня абстракции **ORM** (Object-Relational Mapper – объектно-реляционное отображение), что обеспечивает возможность представления базы данных в виде объектов – в объектно-ориентированных языках программирования.

Базы данных SQL состоят из **таблиц**, где каждая таблица состоит из **строк** (например, записей, элементов, объектов) и **столбцов** (например, атрибутов, полей), напоминая своей организацией электронные таблицы. Платформа Django содержит мощную систему ORM, которая классы представляет как таблицы, объект – как отдельные строки внутри этих таблиц, а атрибуты объектов – как столбцы таблиц.

Представление: шаблоны отображения в разметку HTML и в другие форматы

Последний вопрос, на который необходимо ответить в процессе разработки веб-приложений – как представить, или отформатировать, информацию, запрошенную и/или возвращаемую посредством протокола HTTP и полученную из базы данных. Обычно эта информация возвращается в формате **HTML** (HyperText Markup Language – язык разметки гипертекста) или в более новом XML-подобном формате **XHTML**, в сопровождении программного кода на языке семейства **JavaScript**, обеспечивающего динамические возможности на стороне броузера, и стилей **CSS** (Cascading Style Sheets – каскадные таблицы стилей) – для обеспечения стилизации визуального представления. Кроме того, в современных приложениях для представления динамических данных используется формат **JSON** («легковесный» формат представления данных) или **XML**.

Для представления данных во многих веб-платформах используется **язык шаблонов**, который смешивает теги HTML с синтаксическими конструкциями, напоминающими язык программирования, позволяющими выполнять итерации через коллекции объектов, логические

операции и многое другое, чтобы обеспечить желаемое динамическое поведение. Простейшим примером мог бы служить статический документ HTML, в который включен элемент логики, отображающий, например, имя текущего зарегистрировавшегося пользователя или ссылку «Login» (зарегистрироваться), если пользователь еще не зарегистрировался.

Некоторые системы шаблонов стремятся полностью соответствовать требованиям XHTML, реализуя свои команды в виде атрибутов тегов HTML, благодаря чему получающийся документ может интерпретироваться как обычный документ HTML. Другие более близко имитируют языки программирования, иногда с помощью синтаксиса «альтернативных тегов», когда программные конструкции окружаются специальными символами, чтобы упростить их чтение и синтаксический анализ. Язык шаблонов Django принадлежит к последней группе.

Сложим все вместе

При представлении трех структурных компонентов, лежащих в основе функционирования Всемирной паутины, описанных выше, был опущен один важный аспект: как они взаимодействуют между собой. Как веб-приложение узнает, какой запрос SQL следует выполнить на основе запроса клиента, и как оно узнает, какой шаблон следует использовать для отображения результатов?

В некоторой степени ответ на этот вопрос зависит от используемых инструментов: каждая веб-платформа или язык может предлагать свое решение. Однако между ними больше общего, чем отличий, и хотя в следующих двух разделах описывается подход, используемый платформой Django, многие из рассматриваемых понятий можно обнаружить и в других веб-платформах.

Понимание моделей, представлений и шаблонов

Как вы только что видели, динамические веб-приложения часто делятся на несколько основных компонентов. В этом разделе мы расширим эти понятия, обсудим привлекаемые методики программирования и рассмотрим, как платформа Django реализует их (с рассмотрением тонкостей и примеров в последующих главах).

Выделение уровней (MVC)

Идея разделения динамических приложений (как веб-приложений, так и других), известная как парадигма **MVC** (Model-View-Controller – модель-представление-контроллер), существует уже достаточно давно и обычно применяется к графическим приложениям на стороне клиента. Как вы уже наверняка догадались, эта парадигма означает, что приложение делится на **модель**, управляющую данными, **представление**,

определяющее, как будут отображаться данные, и **контроллер**, осуществляющий посреднические функции между первыми двумя уровнями и дающий пользователю возможность запрашивать данные и управлять ими.

Разделение приложения таким способом обеспечивает необходимую гибкость и кроме всего прочего позволяет многократно использовать один и тот же программный код. Представьте, что имеется некоторое представление – например, модуль, способный отображать числовые данные в графическом виде, – этот модуль можно было бы использовать для различных наборов данных при условии наличия связующего звена между модулем и данными. Или какой-то один определенный набор данных мог бы быть представлен в различных форматах, таких как вышеупомянутое графическое представление, в виде простого текстового файла или в виде таблицы, поддерживающей возможность сортировки. Набор из нескольких контроллеров можно было бы использовать для обеспечения различных уровней доступа к одной и той же модели данных для разных пользователей или давать возможность ввода данных посредством приложений с графическим интерфейсом, а также через электронную почту или командную строку.

Ключом к успеху использования архитектуры MVC является правильное разделение приложения на эти уровни. Наличие в модели данных информации о том, как эти данные должны отображаться, в некоторых случаях может быть удобно, но, как правило, это усложняет смену одного представления другим. Точно так же наличие кода, обеспечивающего доступ к конкретной базе данных, в коде графического представления, может превратиться в бесконечную головную боль при необходимости перейти на использование другой платформы баз данных!

Подход, используемый в платформе Django

Платформа Django придерживается такого разделения обязанностей, хотя и делает это в несколько иной манере, отличающейся от привычной. Суть модели остается той же самой: на уровне модели в Django реализуется только запись и получение данных из базы. Однако уровень «представления» в Django не является последним этапом в отображении данных – представления в платформе Django по своей сути ближе к «контроллерам» в архитектуре MVC. Они являются функциями на языке Python, которые связывают между собой уровень модели и уровень отображения (состоящий из разметки HTML и языка шаблонов платформы Django, который будет рассматриваться ниже, в главе 6 «Шаблоны и обработка форм»). Процитируем высказывание членов команды разработчиков Django:

> В нашей интерпретации MVC «представление» описывает данные, отображаемые перед пользователем. Важно не то, как данные должны выглядеть, а то, какие данные должны быть представлены. Другими словами, представление описывает, какие

данные вы увидите, а не то, как они будут выглядеть. Это довольно тонкое отличие.

Проще говоря, платформа Django разбивает уровень представления на два – метод представления, определяющий, какие данные из модели будут отображаться, и шаблон, определяющий окончательное представление информации. Что касается контроллера, то его функции выполняются самой платформой – она обладает механизмами, определяющими, какое представление и шаблон должны использоваться в ответ на конкретный запрос.

Модели

Основой любого приложения, будь то веб-приложение или любое другое, является информация, которую это приложение собирает, модифицирует и отображает. С точки зрения многоуровневой архитектуры приложения, модель является самым нижним уровнем, или фундаментом. Представления и шаблоны могут добавляться и удаляться, изменять порядок ввода/вывода данных и их представления, но модель остается практически неизменной.

С точки зрения проектирования многоуровневого веб-приложения, модель является, пожалуй, самой простой для понимания и самой сложной в реализации. Моделирование задач реального мира в объектно-ориентированной системе часто является относительно простой задачей, но с точки зрения веб-сайтов, работающих с высокой нагрузкой, самая реалистичная модель не всегда является самой эффективной.

Модель включает широкий диапазон потенциальных ловушек, одна из которых связана с изменением программного кода модели уже после развертывания приложения. Несмотря на то, что «изменяется всего лишь программный код модели», тем не менее, в действительности изменяется структура базы данных, а это часто отражается на данных, уже хранящихся в базе. В последующих главах мы рассмотрим множество подобных проблем, когда будем исследовать архитектуру некоторых примеров приложений.

Представления

Представления в значительной степени (иногда полностью) формируют логику приложений на платформе Django. Их определение выглядит обманчиво просто: функции на языке Python, связанные с одним или более адресами URL и возвращающие объекты ответов HTTP. Все, что будет происходить между этими двумя точками применения механизмов платформы Django, реализующих работу с протоколом HTTP, целиком и полностью зависит от вас. На практике на этом этапе обычно решается несколько похожих задач, таких как отображение объекта или списка объектов, полученных от модели, или добавление новых объектов, а также проверка аутентификации пользователя при-

ложения и либо предоставление, либо отклонение попытки доступа к данным.

Платформа Django предоставляет множество вспомогательных функций для решения подобных задач, но вы можете реализовать всю логику работы самостоятельно, чтобы иметь полный контроль над процессом, широко использовать вспомогательные функции для быстрого создания прототипа и разработки самого приложения или комбинировать эти два подхода. В ваших руках гибкость и мощь.

Шаблоны

Вы должны были заметить, что мы только что заявили, что *представление* отвечает за отображение объектов, полученных из модели. Это верно не на 100 процентов. Если подразумевать, что методы просто возвращают ответ HTTP, это достаточно верно – можно было бы реализовать на языке Python вывод строки и возвращать ее, и это было бы ничуть не хуже. Однако в подавляющем большинстве случаев такая реализация крайне неэффективна и, как уже упоминалось ранее, очень важно придерживаться деления на уровни.

Вместо этого для отображения результатов в разметку HTML, которая часто является конечным результатом работы веб-приложения, большинство разработчиков приложений на платформе Django используют язык шаблонов. Шаблоны, по сути, являются текстовыми документами в формате HTML, со специальным форматированием там, куда выводятся значения, получаемые динамически, поддерживающие возможность использовать простые логические конструкции, такие как циклы и другие. Когда от представления требуется вернуть документ HTML, оно обычно указывает шаблон, передает в него информацию для отображения и использует отображенный шаблон в своем ответе.

Хотя HTML является наиболее типичным форматом, в действительности шаблоны не обязательно должны содержать какую-либо разметку HTML – шаблоны могут использоваться для воспроизведения любого текстового формата, такого как CSV (comma-separated values – значения, разделенные запятыми), или даже текста сообщения электронной почты. Самое важное состоит в том, что шаблоны позволяют отделить отображение данных от программного кода представления, которое определяет, какие данные следует представить.

Общий обзор архитектуры Django

К настоящему моменту мы рассмотрели некоторые из наиболее крупных архитектурных компонентов, составляющих саму систему Django, а также вспомогательные компоненты, не входящие в ее границы. Давайте теперь сложим их вместе, чтобы получить общее представление. На рис. 3.1 можно видеть, что ближе всего к пользователю располагается протокол HTTP. С помощью адресов URL пользователи могут

отправлять запросы веб-приложениям на платформе Django и принимать ответы посредством своих веб-клиентов, которые могут также выполнять программный код JavaScript и использовать технологию Ajax для взаимодействия с сервером.

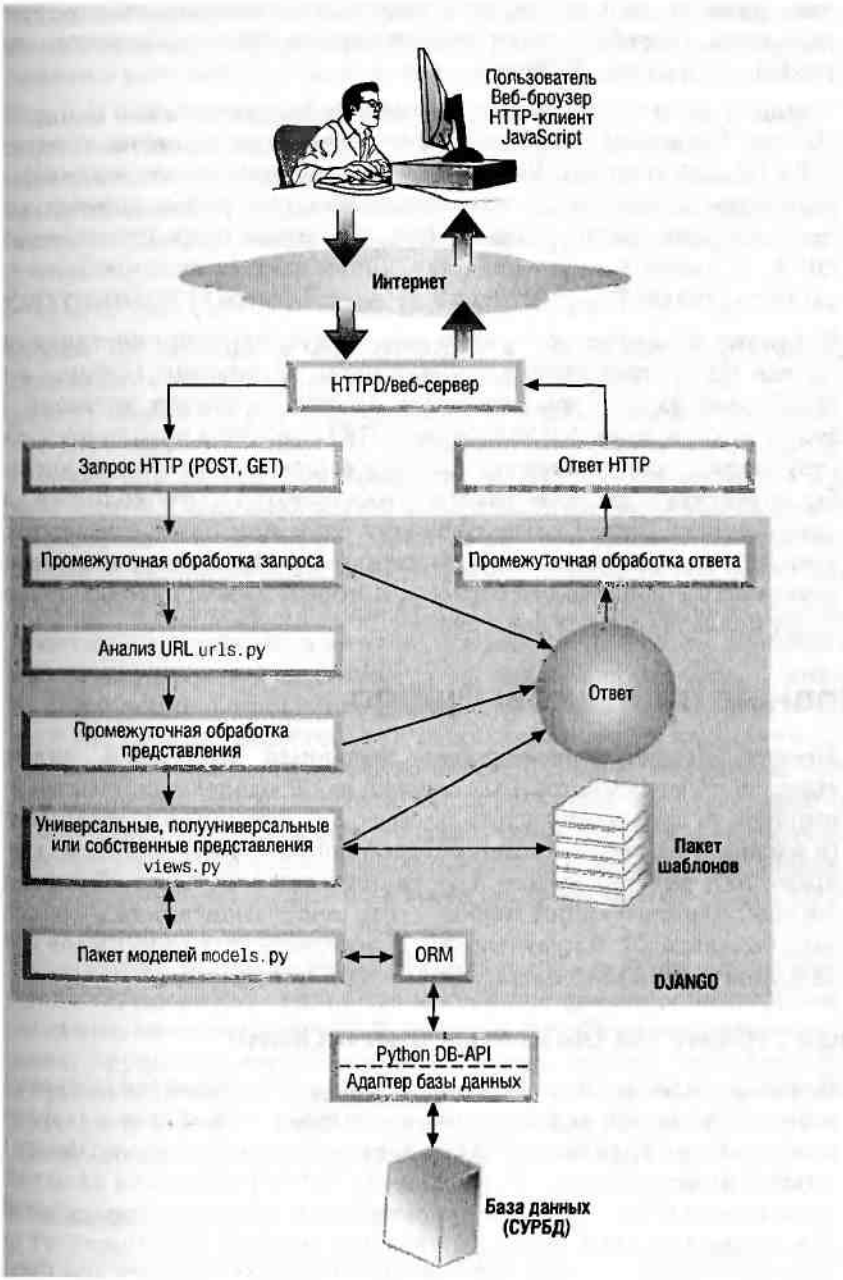

Рис. 3.1. Общая схема архитектуры Django

На другом конце спектра (внизу рисунка) можно видеть базу данных – хранилище информации, которое управляется с помощью моделей и механизма ORM платформы Django, взаимодействующих с базой данных посредством DB-API языка Python и клиентских библиотек базы данных. Эти библиотеки используются в качестве промежуточного звена, они обычно написаны на языке C/C++ и предоставляют интерфейс для языка Python.

Наконец, в середине располагается основа приложения на платформе Django. Парадигма MVC на языке Django превращается в парадигму MTV (Model-Template-View – модель, шаблон, представление). Представления играют роль контроллера между моделью данных, создающей, изменяющей и удаляющей данные в базе посредством механизма ORM, и окончательным представлением данных пользователю с помощью шаблонов.

Соединив все вместе, получаем следующую картину: поступающие запросы HTTP передаются веб-сервером платформе Django, которая принимает их на промежуточном уровне обработки запросов. После этого, исходя из шаблонов адресов URL, запросы передаются соответствующему представлению, которое выполняет основную часть работы, задействуя при этом модель и/или шаблоны, необходимые для создания ответа. Затем ответ проходит один или более промежуточных уровней, где выполняется окончательная обработка перед передачей ответа HTTP обратно веб-серверу, который в свою очередь отправляет ответ пользователю. Улавливаете?

Основные принципы Django

Проект Django, первоначально созданный небольшой, сплоченной группой программистов, был основан на определенных идеалах и принципах и продолжает на них оставаться. Эти идеалы отражают опыт (и в известной степени личности) членов основной команды, и в то же время они выстраиваются в соответствии с тем, что любой веб-разработчик, использующий любой набор инструментальных средств, назвал бы «передовой практикой». Понимание этих принципов поможет вам понять и эффективнее использовать платформу.

Django стремится быть «питонической»

Используемый язык программирования и сообщество пользователей этого языка порой являются одним из самых значительных факторов, влияющих на архитектуру любого проекта, и платформа Django в этом смысле не исключение. Пользователи Python стремятся называть «**питоническим**» все, что хорошо сочетается и в целом придерживается философии языка Python. Для данного термина не существует четкого определения, но обычно это означает, что программный код проявляет признаки, типичные для других работ на языке Python.

Среди этих признаков можно назвать использование краткого, но мощного синтаксиса (синтаксис циклов `for` по умолчанию или еще более краткий синтаксис генераторов списков); идею единственного *правильного* способа решения простой задачи (такие «способы» часто встроены непосредственно в язык программирования, например метод `get` словарей) и предпочтение явного перед неявным (например, обязательный аргумент `self` во всех методах объектов).

Как мы увидим в третьей части «Приложения Django в примерах», многие соглашения, методы и архитектурные решения являются «питоническими» или стремятся быть таковыми. Это помогает опытным программистам быстрее освоить платформу и помогает прививать оптимальные приемы программирования менее опытным разработчикам.

Не повторяйся (Don't Repeat Yourself, DRY)

Одним из признаков «питоничности», заслуживающих отдельного описания, является принцип **DRY** (Don't Repeat Yourself – не повторяйся), общий практически для всех языков программирования. Понятие DRY является, пожалуй, одной из простейших идиом программирования для любого программиста, потому что она является простой, давнишней и разумной идеей: если некоторая информация размещается более чем в одном месте, то, когда вам понадобится изменить ее, вам придется выполнить одну и ту же работу дважды (а то и больше раз).

В качестве примера действия принципа DRY рассмотрим реализацию простых вычислений, в которых участвуют несколько элементов данных, например, вычисление суммы денежных средств, хранящихся на банковских счетах, принадлежащих заданному человеку. В системе с неудачным дизайном эти вычисления могут выполняться в нескольких местах: в странице со списком людей, в странице с подробной информацией о человеке или в странице, вычисляющей общую сумму сразу по нескольким людям. В такой системе, как система ORM платформы Django, легко можно избежать повторений, создав класс `Person` с методом `sum_accounts` и затем использовать его везде, где это потребуется.

Несмотря на то, что идиома DRY легко применяется в простейших ситуациях, таких как только что описанный пример, она является одной из самых тяжелых заповедей, которых трудно придерживаться постоянно, – существует масса ситуаций, когда она вступает в противоречие с другими идиомами и когда ею приходится пожертвовать. Однако эта цель достойна того, чтобы стремиться к ней, и принадлежит к числу целей, которые легче достигаются с опытом.

Слабая зависимость и гибкость

Django – это полноценная веб-платформа, полноценная в том смысле, что включает в себя все компоненты, необходимые для разработки

5 Зак. 1496

динамических веб-приложений: компоненты доступа к базам, платформу запросов, прикладную логику, систему шаблонов и т. д. Однако были потрачены определенные усилия, чтобы не ограничивать пользователям свободу выбора: вы можете задействовать такую долю платформы Django, какая вам потребуется, и заменять ее компоненты другими инструментами, которые, на ваш взгляд, лучше подходят для решения поставленных задач.

Например, некоторым пользователям не нравится система шаблонов Django, и они предпочитают использовать альтернативные решения, такие как Kid и Cheetah. Методы представлений платформы Django не требуют, чтобы использовалась система шаблонов Django, поэтому вполне возможно иметь одно представление, загружающее Kid или Cheetah, отображающее данные в шаблон, написанный в одной их этих систем, и возвращающее результат как часть объекта ответа Django.

То же относится к уровню базы данных. Пользователи, предпочитающие, к примеру, SQLAlchemy или SQLObject, могут просто игнорировать систему ORM платформы Django и работать со своими данными с помощью других инструментов. И наоборот, если пользователь ограничен в доступных средствах, он может использовать в своих проектах (даже не ориентированных на работу во Всемирной паутине) только систему ORM платформы Django, хотя это не так типично.

После всего вышесказанного следует заметить, что такая модульность имеет свою цену: некоторые из замечательных особенностей Django, такие как универсальные методы представлений, позволяющие легко организовать отображение, обновление и создание новых записей в базе данных, охватывают всю платформу целиком. Однако тем, кто еще плохо знаком с особенностями разработки веб-приложений на языке Python, лучше избегать модульного подхода.

Быстрая разработка

Платформа Django была написана для обеспечения быстрой, маневренной разработки. Для работы в быстро изменяющемся новостном веб-сайте основному коллективу требовался комплект инструментов, который позволял бы им воплощать свои идеи в течение чрезвычайно короткого интервала времени. Открытость платформы никак не меняет того, что она является одной из лучших в этой области.

Платформа Django предлагает средства быстрой разработки на нескольких уровнях. Самым очевидным средством является коллекция упоминавшихся ранее универсальных представлений, которая состоит примерно из десятка типичных задач. В комплексе с мощной и гибкой возможностью параметризации веб-сайт может быть полностью основан на этих универсальных представлениях, позволяющих создавать и изменять записи в базе данных, отображать списки объектов (с упорядочением по дате или как-то иначе), отдельные страницы для каждого объекта и многое другое. С помощью всего лишь трех файлов

с программным кодом на языке Python – параметры настройки сайта, объявление модели и карта отображений адресов URL на универсальные представления – и нескольких шаблонов HTML можно создать законченный веб-сайт за несколько минут или часов.

На более низком уровне платформа Django предоставляет множество методов решения типичных задач непосредственно на уровне самого языка Python, поэтому, даже когда универсальные представления не в состоянии удовлетворить все потребности, программист по-прежнему может избежать необходимости писать стереотипный программный код. Среди таких методов имеются методы, отображающие шаблоны со словарями данных, получающие объекты из базы данных и возвращающие ошибку HTTP, если запрошенных объектов не существует, выполняющие обработку форм и т. д.

В соединении с гибкостью, краткостью и выразительностью языка Python эти методы позволяют программистам сосредоточиться на быстром создании и выпуске проекта и/или на решении задач предметной области, не отвлекаясь на грязную работу или на разработку так называемого связующего программного кода.

В заключение

В этой главе мы рассмотрели множество базовых сведений: что такое веб-разработка, какие подходы к созданию веб-сайтов использует платформа Django и подобные ей, и основные принципы, влияющие на разработку самой платформы Django и принятие архитектурных решений. Независимо от того, с каким багажом знаний вы пришли к этой главе, мы надеемся, что вы кое-что получили от этого краткого обзора.

К этому моменту чтения книги вы должны иметь неплохое представление об основах разработки веб-приложений, а также знать теоретические основы и организацию типичной веб-платформы. Во второй части книги, «Подробно о Django», мы углубимся в изучение вопросов использования платформы Django, займемся исследованием различных классов, функций и структур данных и рассмотрим множество фрагментов программного кода, которые помогут во всем этом разобраться.

II

Подробно о Django

4. Определение и использование моделей
5. Адреса URL, механизмы HTTP и представления
6. Шаблоны и обработка форм

4

Определение и использование моделей

Как уже объяснялось в главе 3, «Начало», модель веб-приложения является обычно его основой и, во всяком случае, отличной отправной точкой в исследовании особенностей разработки на платформе Django. Хотя эта глава поделена на два подраздела – определение моделей и затем их использование – тем не менее, это деление можно считать весьма условным. При создании модели мы должны знать, как она будет использоваться, чтобы определить наиболее эффективную иерархию классов и взаимоотношения между ними. И, конечно же, эффективное использование модели невозможно без полного ее понимания.

Определение моделей

Уровень модели базы данных в платформе Django основан на использовании ORM (Object-Relational Mapper – объектно-реляционное отображение), и будет совсем нелишним разобраться, почему было принято такое архитектурное решение, а также узнать о плюсах и минусах этого подхода. По этой причине данный раздел начинается с описания механизма ORM в платформе Django, после чего мы перейдем к изучению полей модели, особенностей взаимоотношений между классами моделей и использования в модели класса метаданных для определения некоторых специфических особенностей поведения модели или с целью обеспечить дополнительные настройки приложения администрирования Django.

Преимущества ORM

Платформа Django наряду с большинством других современных веб-платформ (и многих других средств разработки) опирается на обширный слой доступа к данным, который служит мостом между реляци-

онной базой данных и объектно-ориентированной природой языка Python. Системы ORM до сих пор остаются предметом жарких дебатов в сообществе разработчиков, которые приводят массу различных аргументов за и против их использования. Платформа Django изначально создавалась в предположении использования системы ORM, поэтому мы представим вам четыре аргумента в ее защиту и в частности, в защиту реализации ORM, входящей в состав Django.

Инкапсуляция методов

Объекты моделей в платформе Django, как будет показано ниже, представляют собой прежде всего способ определения коллекций полей, отображаемых в столбцах таблиц базы данных. Это первый и основной шаг в связывании реляционной базы данных с объектно-ориентированными концепциями. Вместо того чтобы выполнять запрос SQL, такой как `SELECT name FROM authors WHERE id=5`, можно просто запросить объект Author, атрибут `id` которого имеет значение 5, и проверить его атрибут `author.name` – такой интерфейс с данными гораздо ближе к языку Python.

Однако объекты модели могут существенно повысить ценность такого подхода. Система ORM платформы Django позволяет определять произвольные методы экземпляров, что может дать массу положительных эффектов. Например:

- Можно определять комбинации полей, или атрибутов, доступных только для чтения, иногда называемых **сгруппированными данными**, или **вычисляемыми атрибутами**. Например, объект `Order` с атрибутами `count` и `cost` мог бы иметь атрибут `total`, который является произведением двух других. Существенно упрощаются типичные объектно-ориентированные шаблоны проектирования, такие как делегирование.
- Система ORM в платформе Django предоставляет возможность переопределять встроенные методы изменения базы данных, такие как методы сохранения и удаления объектов. Это позволяет легко определять произвольный набор операций, которые будут применяться к данным перед тем, как они будут записаны в базу данных или обеспечат выполнение необходимых заключительных действий после удаления записи независимо от того, где и как это удаление было произведено.
- Интеграция с языком программирования – Python в случае с платформой Django – обеспечивается очень просто, что позволяет вам реализовать свои объекты базы данных в соответствии с определенными интерфейсами.

Переносимость

Благодаря своей природе – будучи слоем программного кода между приложением и самой базой данных – системы ORM обеспечивают вы-

сокую переносимость. Большинство систем ORM поддерживают множество типов баз данных, и ORM платформы Django не является исключением. К моменту написания этих строк программный код, использующий модели Django, мог взаимодействовать с такими базами данных, как PostgreSQL, MySQL, SQLite и Oracle, и этот список будет продолжать расти по мере создания новых модулей для работы с базами данных.

Безопасность

Поскольку при использовании ORM крайне редко приходится выполнять собственные запросы SQL, отпадает необходимость волноваться о проблемах, которые могут быть вызваны неправильно сформированной или слабо защищенной строкой запроса, что часто приводит к таким проблемам, как нападения, производимые за счет внедрения зловредного кода SQL. Кроме того, системы ORM предоставляют централизованный механизм экранирования входных переменных, избавляя программиста от необходимости тратить время на подобные мелочи. Такого рода преимущества характерны для модульного или многоуровневого программного обеспечения, отличными примерами которого могут служить платформы MVC. Когда весь программный код, ответственный за решение задач определенной предметной области, хорошо организован и обособлен, он часто позволяет сэкономить массу времени и обеспечить высокий уровень безопасности.

Выразительность

Несмотря на то, что это не имеет прямого отношения к *определениям* моделей, одним из главных преимуществ использования ORM (и самым большим отличием, в сравнении с созданием запросов SQL) является синтаксис запросов, используемый для извлечения записей из базы данных. Мало того, что синтаксис высокого уровня проще в работе, но сам перенос механизма запросов на язык Python предоставляет возможность использовать полезные методы и принципы. Например, становится возможным конструировать сложные запросы, выполняя обход структур данных, причем реализация такого решения получается более компактной, чем эквивалентный запрос SQL; к тому же при этом удается избежать необходимости утомительного манипулирования строками, которая возникает в противном случае.

Богатый набор типов полей в Django

Модели платформы Django обладают широким диапазоном типов полей. Некоторые из них тесно связаны со своими реализациями в базе данных, другие создавались с учетом особенностей интерфейса веб-форм. Однако большая их часть занимает промежуточное положение. Хотя исчерпывающий перечень полей можно найти в официальной документации к платформе Django, мы представим сравнительный

обзор некоторых из наиболее часто используемых типов. Но сначала дадим краткое введение в основы определения моделей в платформе Django.

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
    length = models.IntegerField()
```

Суть этого фрагмента программного кода должна быть очевидна – это упрощенная модель книги, соединяющая в себе некоторые понятия, связанные с базой данных. Модель содержит не очень много информации – вообще говоря, тех, кто занимается каталогизацией книг, интересует намного больше, чем просто название, автор и количество страниц, – но в качестве примера годится. Кроме того, она справляется со своей задачей. Этот пример можно было бы поместить в файл models.py и получить возможность создать приложение книжного каталога, добавив совсем немного.

Как видите, для представления объектов платформа Django использует классы, написанные на языке Python, обычно отображаемые в таблицы базы данных, атрибуты которых отображаются в столбцы. Эти атрибуты сами по себе являются объектами, а точнее подклассами родительского класса Field. Как уже говорилось выше, некоторые из них являются прямыми аналогами типов столбцов SQL, а другие обеспечивают некоторый уровень абстракции. Исследуем некоторые подклассы класса Field.

- CharField и TextField: это, пожалуй, наиболее часто используемые типы полей. Оба типа преследуют практически одну и ту же цель – хранение текста. Разница лишь в том, что поле типа CharField имеет ограниченную длину, а поле типа TextField – практически неограниченную. Какой из двух типов использовать, зависит от ваших потребностей, включая потребность выполнять полнотекстовый поиск в базе данных или потребность обеспечить высокую эффективность хранения.
- EmailField, URLField и IPAddressField: эти три типа полей также по сути относятся к типу CharField, но снабженному дополнительными возможностями проверки. Значения этих типов сохраняются в базе данных как CharField, но реализуют программный код, выполняющий проверку, гарантирующую, что содержимое будет соответствовать требованиям, предъявляемым к адресам электронной почты, адресам URL и IP-адресам соответственно. Точно так же легко можно создавать свои собственные «типы полей», добавляя необходимые проверки к полям моделей, на том же уровне, где находятся типы полей, встроенные в платформу Django. (Дополнительную ин-

формацию о проверке значений можно найти в главе 6 «Шаблоны и обработка форм» и в главе 7 «Фотогалерея».)

- BooleanField и NullBooleanField: тип BooleanField может использоваться везде, где требуется сохранять значение True или False, но иногда бывает необходимо иметь возможность помимо этих двух значений сохранять еще и третье значение, имеющее смысл *не известно* – в этих случаях поле должно оставаться пустым или иметь значение NULL, для чего используется тип NullBooleanField. Различия между этими двумя полями подчеркивают, что моделирование данных требует некоторых размышлений и что решения иногда должны приниматься не только на техническом, но и на семантическом уровне, то есть необходимо учитывать не только, как хранить данные, но и какую смысловую нагрузку они несут.
- FileField: этот тип является одним из самых сложных типов полей, причем в значительной степени потому, что практически вся его работа связана не с базой данных, а с обработкой запроса в платформе. Поле FileField хранит в базе данных только путь к файлу – подобно родственному типу FilePathField, но в отличие от него идет немного дальше и обеспечивает возможность выгрузки файла из броузера пользователя и сохранения его где-то на сервере. Кроме того, этот тип предоставляет методы доступа к выгруженному файлу посредством веб-адреса URL.

Это лишь часть типов полей, которые могут использоваться в определениях моделей Django, и с выходом новых версий к ним могут добавляться новые поля или расширяться возможности существующих. Актуальный перечень всех классов полей моделей и их описание можно найти в официальной документации Django. Кроме того, многие из этих полей вы увидите в этой книге, в примерах программного кода и в примерах приложений в третьей части «Приложения Django в примерах».

Первичные ключи и уникальные значения

Одним из широко известных понятий в определении реляционных баз данных является понятие **первичного ключа**. Если поле определено как первичный ключ, тем самым гарантируется уникальность значений этого поля во всей таблице (или, если говорить в терминах ORM, во всей модели). Обычно в качестве первичных ключей используются целочисленные автоинкрементные поля, потому что такие поля предоставляют простой и эффективный способ обеспечения уникальности каждой строки в таблице.

Их также удобно использовать в качестве ссылок при описании взаимоотношений между моделями (о которых будет рассказываться в следующем разделе). Например, если некоторый объект Book имеет идентификационный номер 5 и при этом гарантируется, что *только один*

объект Book может иметь этот идентификационный номер, то ссылка «книга №5» будет однозначно указывать на нужную книгу.

Поскольку такой тип первичного ключа используется достаточно часто, платформа Django автоматически создает его по умолчанию, если не указано обратное. Все модели, в которых отсутствует явное определение первичного ключа, получают атрибут id типа AutoField (автоинкрементное целочисленное поле). Тип AutoField ведет себя как обычное целое число, а поведение лежащего в его основе типа столбца зависит от используемой базы данных.

Тем, кому требуется больший контроль над первичными ключами, достаточно просто указать primary_key=True для одного из полей модели. В результате вместо поля id (которое в этом случае не создается) первичным ключом таблицы станет это поле. Это означает, что значения поля должны быть полностью уникальными, поэтому нельзя использовать в качестве первичных ключей строковые поля, предназначенные для хранения имен или других идентификаторов, если вы на 110 процентов не уверены, что среди них не будут встречаться повторяющиеся значения.

Говоря о дубликатах, следует также упомянуть о существовании похожего аргумента, который может применяться к определениям любых полей модели: unique=True. Этот аргумент принудительно обеспечивает уникальность значений поля и при этом не делает его первичным ключом.

Отношения между моделями

Способность определять отношения между объектами моделей является одной из самых сильных особенностей, обусловленных использованием реляционных баз данных (где слово *реляционные* переводится как *отношения*), а также одной из областей, где различные системы ORM могут отличаться друг от друга. Текущая реализация Django в значительной степени опирается на возможности баз данных и гарантирует, что отношения между данными определяются не на уровне приложения, а на уровне базы данных. Однако, так как SQL предусматривает только одну явную форму отношений – внешний ключ, для обеспечения более сложных отношений необходимо добавить некоторое иерархическое представление. Сначала мы исследуем сам внешний ключ, а затем рассмотрим, как его можно использовать для построения более сложных типов отношений.

Внешние ключи

Простота внешних ключей обусловливает простоту их реализации в платформе Django. Внешние ключи представлены собственным подклассом ForeignKey класса Field. При создании поля внешнего ключа первым аргументом ему передается класс модели, на который он будет ссылаться, как показано в примере ниже:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
```

Следует заметить, что класс, на который будет ссылаться внешний ключ, должен быть предварительно объявлен, в противном случае имя Author будет недоступно для использования в поле типа ForeignKey класса Book. Однако имеется возможность указывать имя класса в виде строки, если имя класса определяется ниже в этом же модуле, или использовать точечную нотацию (например, 'myapp.Author') в противном случае. Ниже приводится измененная версия предыдущего фрагмента, где методу ForeignKey передается имя класса в виде строки:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey("Author")

class Author(models.Model):
    name = models.CharField(max_length=100)
```

Кроме того, имеется возможность определять внешние ключи, ссылающиеся на собственный класс, для чего в методе ForeignKey следует использовать строку 'self'. Такие ключи часто используются при определении иерархических структур (например, класс Container определяет атрибут parent, за счет чего допускается вкладывать объекты типа Container друг в друга) или в похожих ситуациях (таких как атрибуты supervisor или hired_by в классе Employee).[1]

Хотя внешний ключ определяется только с одной стороны отношения, тем не менее, принимающая сторона оказывается в состоянии следовать установленному отношению. С технической точки зрения внешние ключи описывают отношение *многие-к-одному*, когда на один и тот же «родительский» объект могут ссылаться несколько «дочерних» объектов, благодаря этому дочерние объекты обладают единственной ссылкой на родительский объект, но родительский объект получает доступ к *множеству* дочерних объектов. Используя определения классов из предыдущего примера, можно было бы использовать экземпляры Book и Author, как показано ниже:

```
# Извлечь книгу из хранилища - порядок выполнения запросов описывается ниже
book = Book.objects.get(title="Moby Dick")
# Определить имя автора - это просто
author = Book.author
# Получить список книг, написанных данным автором
books = author.book_set.all()
```

[1] Employee — наемный работник, supervisor — администратор, hired_by — наниматель. — *Прим. перев.*

Как видите, «обратная связь» от Author к Book представлена атрибутом Author.book_set (объект-менеджер, который будет описан ниже в этой главе), который автоматически добавляется механизмом ORM. Существует возможность изменить такой порядок именования, для чего при вызове метода ForeignKey необходимо указать аргумент related_name. В предыдущем примере мы могли бы определить атрибут author как ForeignKey("Author", related_name="books") и обращаться к списку книг author.books вместо author.book_set.

Примечание

Аргумент related_name можно не использовать в случае простых иерархий объектов, но необходимо в случае более сложных иерархий, например, когда имеется несколько внешних ключей, ведущих из одного объекта в другой. В таких ситуациях механизму ORM нужно сообщить, как отличать два менеджера обратных связей на стороне, куда ссылаются два поля ForeignKey. Если вы забудете об этом, инструменты баз данных, входящие в состав Django, напомнят вам об этом с помощью сообщений об ошибках!

Отношения многие-к-многим

Внешние ключи обычно используются для организации отношений **один-к-многим** (или **многие-к-одному**) – в предыдущем примере книга могла иметь единственного автора, а автору могло принадлежать множество книг. Однако иногда бывает необходимо обеспечить большую гибкость. Например, до сих пор мы предполагали, что у книги может быть только один автор, но как быть с книгами, которые написаны несколькими авторами, как, например, эта книга?

В таких случаях отношения типа «к многим» могут складываться не только с одной стороны (автор может написать одну или более книг), но и с другой (книга тоже может быть написана одним или более авторами). Для решения таких задач используются отношения **многие-к-многим**, но, так как на языке SQL невозможно определять подобные отношения, мы будем строить их с помощью внешних ключей.

Для подобных ситуаций платформа Django предоставляет еще один тип полей, описывающий такой тип отношений: ManyToManyField. Синтаксически этот тип идентичен ForeignKey – вы определяете поле данного типа с одной стороны отношения, указывая класс, находящийся на другой стороне, а механизм ORM автоматически добавляет к другой стороне необходимые методы или атрибуты, обеспечивающие возможность использовать отношение (обычно за счет создания менеджера _set, как в случае использования ForeignKey). Однако вследствие природы отношения многие-к-многим совершенно неважно, с какой стороны оно будет определено, потому что по своей сути этот тип отношений является симметричным.

Примечание

Если вы предполагаете использовать приложение администрирования из платформы Django, имейте в виду, что в форме объектов, имеющих отношения «многие-к-многим», отображаются только поля на стороне *определения* отношения.

Примечание

Поля ManyToManyField, ссылающиеся на свой класс (то есть, когда поле ManyToManyField модели ссылается на свою собственную модель), являются симметричными по умолчанию, потому что предполагается, что в отношение вовлечены обе стороны. Однако это не всегда верно, и поэтому предоставляется возможность изменить такое поведение, определив аргумент symmetrical=False.

Дополним наш пример с книгами новой реализацией, обеспечив возможность указывать несколько авторов одной книги:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
```

Использование отношения, устанавливаемого полем ManyToManyField, напоминает использование отношения первичного ключа на стороне «ко многим»:

```
# Извлечь книгу из хранилища
book = Book.objects.get(title="Python Web Development Django")
# Определить имена авторов
authors = Book.author_set.all()
# Получить список книг, написанных третьим автором
books = authors[2].book_set.all()
```

Секрет полей типа ManyToManyField заключается в том, что за кулисами они создают совершенно новую таблицу, которая обеспечивает возможность поиска, необходимую при таком типе отношений, а в этой таблице используются внешние ключи SQL – каждая строка представляет единственное отношение между двумя объектами и содержит внешние ключи на оба объекта.

При обычном использовании механизма ORM платформы Django эта поисковая таблица скрыта и к ней нельзя обратиться иначе, как через одну из сторон отношения. Однако существует возможность определить специальный параметр для поля ManyToManyField – through, в котором явно определить класс промежуточной модели. Использование параметра through позволяет вам самостоятельно управлять дополнительными полями в промежуточном классе, сохраняя удобство использования менеджеров с обеих сторон отношения.

Ниже приводится пример, идентичный предыдущему, где используется поле ManyToManyField, но содержащий явное определение промежуточной таблицы Authoring, которая добавляет к отношению дополнительное поле collaboration_type, и ключевое слово through, указывающее на нее.

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author, through="Authoring")

class Authoring(models.Model):
    collaboration_type = models.CharField(max_length=100)
    book = models.ForeignKey(Book)
    author = models.ForeignKey(Author)
```

Вы можете обращаться к моделям Author и Book тем же способом, что и в предыдущем примере, а кроме того, получаете возможность конструировать запросы, определяющие жанр творчества автора.

```
# Получить подборку всех очерков, в создании которых участвовал Chun
chun_essay_compilations = Book.objects.filter(
    author__name__endswith='Chun',
    authoring__collaboration_type='essays'
)
```

Как видите, эта возможность повышает гибкость Django в составлении отношений, имеющих реальное наполнение.

Составление отношений один-к-одному

В дополнение к отношениям многие-к-одному и многие-к-многим, которые только что были рассмотрены, в разработке реляционных баз данных иногда используется третий тип отношений – **один-к-одному**. Как и в двух первых случаях, название этого типа отношений говорит само за себя – с обеих сторон отношения существует только один объект, вовлеченный в отношение.

Платформа Django реализует эту концепцию в виде типа OneToOneField, который во многом идентичен типу ForeignKey – он требует единственный аргумент, класс, с которым устанавливается отношение (или строка «self», когда модель устанавливает отношение с самой собой). Кроме того, при создании поля этого типа имеется возможность передавать необязательный аргумент related_name, чтобы обеспечить отличия между несколькими однотипными отношениями, устанавливаемыми между одними и теми же двумя классами. В отличие от родственных типов, создание поля OneToOneField не приводит к созданию менеджера обратной связи – только еще один обычный атрибут, потому что в любом направлении отношения всегда существует только один объект.

Этот тип отношений чаще всего используется для поддержания состава объектов, или владения, и поэтому чаще связан с объектно-ориентированной архитектурой, чем с отношениями в реальном мире. До появления в Django поддержки прямого наследования моделей (смотрите ниже) поля OneToOneField обычно использовались для организации отношений наследования и продолжают составлять скрытую от глаз основу для этой особенности.

Ограничение отношений

Последнее замечание относительно определения отношений: для ForeignKeys и ManyToManyField существует возможность определять аргумент limit_choices_to. В этом аргументе передается словарь, в котором пары ключ/значение являются ключевыми словами и значениями (что собой представляют ключевые слова – описывается ниже). Это мощный способ определить возможные значения определяемых вами отношений.

Например, ниже приводится версия класса модели Book, работающая только с авторами, чьи имена заканчиваются на «Smith»:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

Примечание

Кроме того, возможно, а иногда и желательно, определить ограничения на уровне формы. Смотрите описание ModelChoiceField и ModelMultipleChoiceField в главе 6.

Наследование модели

Возможность наследования моделей – это относительно новая (на момент написания этой книги) особенность ORM платформы Django. В дополнение к внешним ключам и другим типам отношений между независимыми в остальном классами моделей имеется возможность задавать модели, одни из которых наследуют другим, – точно так же, как это делается в обычных классах на языке Python, не имеющих отношения к ORM. (Несколько примеров можно найти в главе 1 «Практическое введение в Python для Django».)

Например, предыдущий класс SmithBook можно было бы определить не как самостоятельный класс, в котором по воле случая оказались те же два поля, что и в классе Book, но явно унаследовать в нем класс Book. Преимущества такого подхода, как нам кажется, вполне очевидны – подкласс, вместо того чтобы полностью копировать определение дру-

гого класса, может просто добавлять или переопределять только те поля, которые отличаются от аналогичных полей в родительском классе.

В нашем упрощенном примере с классом Book эти преимущества не выглядят достаточно внушительно, но представьте себе более реалистичную модель с десятком, а то и более, атрибутов и с несколькими сложными методами, – тогда наследование превратится в весьма привлекательный способ следования принципу «Не повторяйся» (Don't Repeat Yourself, DRY). Однако следует заметить, что метод композиции, выражающийся в использовании ForeignKey или OneToOneField, остается вполне приемлемой альтернативой! Какой прием использовать, полностью зависит от вас и от разрабатываемой вами модели.

В настоящее время платформа Django реализует два различных подхода к наследованию, каждый из которых имеет свои преимущества и недостатки: **абстрактные базовые классы** и **многотабличное наследование**.

Абстрактные базовые классы

Наследование с использование абстрактных базовых классов – это самое обычное наследование, реализованное в языке Python. Этот прием позволяет реструктурировать определения моделей так, чтобы общие поля и методы наследовать от базовых классов. Однако на уровне запросов к базе данных базовых классов не существует, и потому их поля копируются в таблицы дочерних классов.

Это выглядит как нарушение принципа «Не повторяйся», но в действительности это может оказаться требуемым поведением – в случае, если *нежелательно* создавать для базового класса дополнительную таблицу в базе данных, например когда база данных уже существует или используется другим приложением. Кроме того, это просто неплохой способ подчеркнуть структуру определений классов, не подразумевая при этом фактическую иерархию объектов.

Рассмотрим повторно (и дополним) иерархию моделей Book и SmithBook, задействовав абстрактные базовые классы:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

    class Meta:
        abstract = True
```

```
class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

Ключевой здесь является инструкция abstract = True внутри класса Meta, вложенного в класс Book, – она показывает, что класс Book является абстрактным и существует только, чтобы обеспечить наличие атрибутов в наследующих его подклассах. Обратите внимание, что класс SmithBook, наследуя класс Book вместо привычного класса models.Model, переопределяет только поле authors, чтобы указать параметр limit_choices_to; в результате в получившейся схеме базы данных присутствуют столбцы title, genre и num_pages, а также таблица поиска author реализации отношения многие-к-многим. Кроме того, на уровне языка Python класс SmithBook обладает унаследованным от класса Book методом __unicode__, который просто возвращает значение поля title.

Другими словами, при создании таблицы в базе данных, а также при использовании для создания объектов, запросов ORM и т. д. класс SmithBook ведет себя так, как если бы он имел следующее определение:

```
class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })

    def __unicode__(self):
        return self.title
```

Как уже упоминалось выше, это поведение распространяется на механизм запросов, а также на атрибуты экземпляров класса SmithBook, поэтому следующий запрос будет вполне допустимым:

```
smith_fiction_books = SmithBook.objects.filter(genre='Fiction')
```

Однако наш пример не совсем подходит для использования абстрактных базовых классов; вы, скорее всего, создали бы самые обычные модели Books и SmithBooks. Абстрактные классы потому и называются абстрактными, что невозможно создавать экземпляры таких классов и, как говорилось выше, они полезны в основном только для поддержки принципа «Не повторяйся» на уровне определения моделей. Для нашего конкретного случая больше подходит многотабличное наследование, которое описывается далее.

Несколько последних замечаний относительно абстрактных базовых классов: вложенный подкласс Meta, присутствующий в подклассах, полностью наследуется от родительского класса (за естественным исключением атрибута abstract, значение которого по умолчанию уста-

навливается равным False, а также некоторых параметров, таких как db_name, имеющих отношение к базе данных).

Кроме того, если в базовом классе при определении поля отношения, такого как ForeignKey, используется аргумент related_name, то необходимо использовать операцию форматирования, чтобы в подклассах не возникало конфликтов имен. Используйте не обычные строки, такие как "related_employees", а добавляйте в них конструкцию %(class), например "related_%(class)s" (если вы не помните этот прием подстановки строк, вернитесь к главе 1). Благодаря этому в строку будет подставляться имя подкласса и конфликтов удастся избежать.

Многотабличное наследование

Многотабличное наследование на уровне определения лишь незначительно отличается от абстрактных базовых классов. Здесь по-прежнему используется наследование классов на уровне языка Python, за исключением того, что в классе Meta отсутствует определение abstract = True. Если исследовать экземпляры моделей или запросы, многотабличное наследование проявляет те же особенности, что мы уже видели прежде – подкласс наследует все атрибуты и методы родительского класса (за исключением класса Meta, по причинам, которые описываются чуть ниже).

Основное отличие заключается в механизмах, лежащих в основе. Родительские классы в этом случае являются полноценными моделями Django со своими собственными таблицами в базе данных; они могут использоваться для создания экземпляров, а также позволяют подклассам наследовать свои атрибуты. Этот тип наследования устанавливается посредством неявного определения отношения один-к-одному между подклассами и родительским классом и выполнением некоторых скрытых действий, связывающих два объекта, благодаря чему дочерний класс наследует атрибуты родительского класса.

Другими словами, многотабличное наследование – это лишь удобная обертка вокруг обычного отношения «имеет», которое также известно, как прием композиции объектов. Так как платформа Django стремится быть «питонической», «скрытое» отношение становится явным, когда в этом возникает необходимость, посредством поля OneToOneField, которому присваивается имя, состоящее из имени родительского класса, в котором все символы приведены к нижнему регистру, с добавлением суффикса _ptr. Например, в следующем фрагменте класс SmithBook получает атрибут book_ptr, связанный с «родительским» экземпляром класс Book.

Ниже приводится версия нашего примера моделей Book и SmithBook с использованием многотабличного наследования:

```
class Author(models.Model):
    name = models.CharField(max_length=100)
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

Как уже говорилось, единственное отличие здесь заключается в отсутствии параметра abstract в определении класса Meta. Если теперь выполнить команду manage.py syncdb с пустой базой данных и с данным файлом models.py, то будут созданы три основные таблицы, по одной для классов Author, Book и SmithBook, тогда как при использовании абстрактного базового класса было бы создано всего две таблицы — Author и SmithBook.

Обратите внимание, что экземпляры класса SmitBook получат атрибуты book_ptr, связанные со скомпонованными с ними экземплярами класса Book, а экземпляры класса Book, принадлежащие (или являющие частью, в зависимости от точки зрения) экземплярам SmithBook, получат атрибуты smithbook (без суффикса _ptr).

Так как эта форма наследования позволяет создавать экземпляры родительского класса, наследование вложенного класса Meta может порождать проблемы или конфликты между двумя сторонами отношения. Вследствие этого необходимо переопределить большую часть параметров в классе Meta, чтобы исключить возможность их совместного использования обоими классами (хотя параметры ordering и get_latest_by наследуются, если не определяются в дочернем классе). Это усложняет следование принципу «Не повторяйся», но достигнуть 100-процентного соответствия этому принципу удается не всегда.

Наконец, мы полагаем, что совершенно ясно, почему этот подход лучше применительно к нашей модели книги — мы имеем возможность создавать обычные объекты обоих классов Book и SmithBook. Если вы используете наследование моделей для отражения отношений реального мира, то, скорее всего, вместо абстрактных базовых классов вы предпочтете использовать многотабличное наследование. Понимание, когда и какой подход использовать и когда не использовать ни один из них, приходит с опытом.

Вложенный класс Meta

Поля и отношения, определяемые в моделях, обусловливают структуру базы данных и имена переменных, которые позднее будут использо-

ваться для обращений к модели данных – вам часто придется добавлять такие методы, как __unicode__ и get_absolute_url, или переопределять встроенные методы save и delete. Однако существует третий важный аспект, связанный с определением модели, – вложенный класс, информирующий платформу Django о наличии различных метаданных, касающихся модели, о которой идет речь: класс Meta.

Класс Meta, как следует из его имени, содержит метаданные модели, описывающие порядок ее использования или отображения: имя модели при ссылке на единственный объект или на множество объектов, порядок сортировки при выполнении запросов к таблице в базе данных, имя таблицы в базе данных (если оно точно известно) и т. д. Кроме того, класс Meta используется для наложения ограничений уникальности сразу по нескольким полям, потому что такие ограничения невозможно наложить при объявлении любого единственного поля. Попробуем добавить некоторые метаданные в наш первый пример модели Book:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

    class Meta:
        # В алфавитном порядке
        ordering = ['title']
```

Вот и все! Класс Book настолько прост, что нет необходимости определять большое число параметров внутри вложенного класса Meta, и если бы нас не волновал порядок сортировки, определение этого класса вообще можно было бы опустить. Классы Meta и Admin вообще не являются обязательными аспектами объявления модели, хотя часто используются. Теперь на скорую руку создадим более сложный пример, потому что метапараметры класса Book не представляют никакого интереса.

```
class Person(models.Model):
    first = models.CharField(max_length=100)
    last = models.CharField(max_length=100)
    middle = models.CharField(max_length=100, blank=True)

    class Meta:
        # Корректный способ определить порядок сортировки имен,
        # предполагается вывод в формате Фамилия, Имя О.
        ordering = ['last', 'first', 'middle']

        # Здесь мы указываем, что не может быть двух людей, имена
        # которых совпадают на все 100%. Конечно, в реальной жизни
        # такое возможно, но мы имитируем идеальный мир.
        unique_together = ['first', 'last', 'middle']

        # По умолчанию в Django множественное число образуется
        # простым добавлением 's' в конце; но в данном случае
        # этот прием не подходит.
        verbose_name_plural = "people"
```

Как следует из комментариев, модель отображения данных человека была бы слишком неконкретной без некоторых определений параметров в классе Meta. Мы использовали все три поля, чтобы определить порядок сортировки записей, избежать дублирования имен и вынудить систему использовать для обозначения нескольких человек слово «people», так как использование слова «persons», хоть и привлекает своей необычностью, но нежелательно.[1]

Описание дополнительных подробностей различных параметров, которые можно определить в классе Meta, вы найдете в официальной документации к платформе Django.

Регистрация в приложении администрирования и дополнительные параметры

Если вы используете приложение администрирования, поставляемое в составе платформы Django, значит, вы интенсивно используете объекты site и их функции register, а также необязательные подклассы ModelAdmin. Эти подклассы позволяют определять различные параметры, касающиеся порядка использования вашей модели при взаимодействии с ней в приложении администрирования.

Простой регистрации класса модели в приложении администрирования (наряду с активацией самого приложения Admin, как описывалось в главе 2 «Django для нетерпеливых: создание блога») вполне достаточно, чтобы это приложение могло работать с моделью и воспроизводить простые страницы со списком и с формой. А в комбинации с подклассом ModelAdmin, определяющим дополнительные параметры, появляется возможность выбирать поля для отображения в списках, состав полей в формах и многое другое.

Кроме того, за счет создания подклассов Inline и использования их в подклассе ModelAdmin допускается определять возможность редактирования параметров полей, устанавливающих отношения, таких как ForeignKey. Такое быстрое увеличение числа дополнительных классов поначалу может показаться странным, но они дают чрезвычайно гибкий способ представления данной модели в нескольких вариантах или в нескольких административных разделах. Расширение иерархии моделей за счет встроенной возможности редактирования также позволяет помещать встроенные формы в более чем одну страницу «родительской» модели, если это необходимо.

Детальное описание каждого параметра смотрите в официальной документации; кроме того, обратите внимание на некоторые примеры использования приложения администрирования в третьей части книги; ниже схематически описываются возможности двух основных типов параметров класса ModelAdmin.

---

1 «People» – люди, «persons» – лица. – *Прим. перев.*

- Форматирование списков: list_display, list_display_links, list_filter и подобные им параметры позволяют менять состав полей, отображаемых в списке (по умолчанию экземпляры модели, будучи простым строковым представлением, отображаются в единственном столбце), а также отыскивать поля и фильтровать ссылки, благодаря чему появляется возможность быстрой навигации по данным.
- Отображение формы: параметры fields, js, save_on_top и другие обеспечивают гибкие средства переопределения формы представления модели по умолчанию, а также позволяют добавлять собственный программный код JavaScript и классы CSS, что бывает удобно, когда появляется желание попытаться вручную изменить внешнее представление в приложении администрирования, чтобы обеспечить визуальное соответствие с остальной частью веб-сайта.

Наконец, вам может показаться, что *слишком* интенсивное использование этих параметров является сигналом к отказу от использования приложения администрирования и написанию своих собственных форм для администратора. Однако прежде чем приступать к этому, обязательно прочитайте раздел «Настройка приложения администрирования» в главе 11 «Передовые приемы программирования в Django», где вы узнаете, насколько гибким может быть приложение администрирования, встроенное в платформу Django.

Использование моделей

Теперь, когда мы выяснили, как определять и расширять модели, рассмотрим, как они создаются и как выполняются запросы к базе данных, а закончим пояснениями о запросах SQL, лежащих в основе всего механизма.

Создание и изменение базы данных с помощью утилиты manage.py

Как отмечалось в главе 2, сценарий manage.py, создаваемый отдельно для каждого проекта, включает функциональность, необходимую для работы с базой данных. Наиболее часто используемой командой сценария manage.py является команда syncdb. Пусть вас не смущает название команды – она не выполняет полную синхронизацию базы данных с вашими моделями, как полагают некоторые пользователи. Вместо этого она гарантирует, что все классы моделей будут представлены таблицами в базе данных с созданием новых таблиц по мере необходимости, но без внесения изменений в существующие таблицы.

То есть, если вы создали некоторую модель и перенесли ее в базу данных с помощью команды syncdb, то при последующих изменениях в модели команда syncdb не будет пытаться синхронизировать базу данных в соответствии с этими изменениями. Предполагается, что

«Синхронизация» базы данных

Такое поведение команды syncdb обусловлено твердым убеждением разработчиков, что данные рабочей версии приложения никогда не должны отдаваться на произвол автоматизированного процесса. Более того, в основе стратегии лежит убеждение, что изменение структуры базы данных должно производиться, только когда разработчик владеет языком SQL в достаточной степени, чтобы выполнить изменения вручную. Авторы книги во многом согласны с таким подходом – при разработке с применением высокоуровневых средств совсем нелишним будет освоить и технологии, лежащие в их основе.

В то же время механизм автоматического или полуавтоматического изменения структуры базы данных (например, при миграции с платформы Rails) часто может ускорить процесс разработки. К моменту написания этих строк существовало несколько проектов, не входящих в ядро платформы Django и находящихся на разных стадиях готовности, в которых предпринимаются попытки восполнить этот недостаток функциональности в платформе.

разработчик должен выполнять такие изменения вручную, с помощью сценариев или просто за счет удаления таблицы или всей базы данных и повторного запуска команды syncdb, что в результате приведет к созданию таблицы или базы данных с учетом всех изменений. Но пока для вас важно понимать, что команда syncdb является основным методом превращения класса модели в таблицу или таблицы базы данных.

Помимо команды syncdb утилита manage.py предоставляет еще несколько функций для работы с базами данных, на которые опирается команда syncdb при выполнении своей работы. В табл. 4.1 приводятся некоторые из наиболее часто используемых функций. Среди них присутствуют команды sql и sqlall, которые отображают инструкции CREATE TABLE (команда sqlall кроме того выполняет загрузку начальных данных); команда sqlindexes создает индексы; команды sqlreset и sqlclear очищают или сбрасывают ранее созданные таблицы; команда sqlcustom выполняет нестандартные начальные инструкции SQL (подробнее об этом рассказывается ниже) и т. д.

В отличие от syncdb команды sql* не изменяют базу данных. Вместо этого они просто выводят инструкции SQL, давая разработчику возможность проверить их (например, чтобы разработчик убедился, что команда syncdb делает именно то, что он ожидает) или сохранить в виде отдельного файла сценария SQL.

Таблица 4.1. Функции manage.py

Функция manage.py	Описание
syncdb	Создает таблицы, необходимые для всех приложений
sql	Выводит инструкции CREATE TABLE
sqlall	То же, что и команда sql, плюс выводит начальные запросы из файла .sql
sqlindexes	Выводит инструкции, создающие индексы для столбцов первичного ключа
sqlclear	Выводит инструкцию DROP TABLE
sqlreset	Комбинация команд sqlclear и sql(DROP плюс CREATE)
sqlcustom	Выводит инструкции SQL из файла .sql
loaddata	Загружает начальную оснастку (напоминает команду sqlcustom, но без кода SQL)
dumpdata	Выводит текущее содержимое базы данных в форматах JSON, XML и других

Кроме того, имеется возможность передать вывод этих команд непосредственно клиенту базы данных для немедленного выполнения – в этом случае они могут действовать, как частичные аналоги команды syncdb. Можно также использовать сразу оба способа – вывести инструкции SQL в файл, отредактировать его, а затем передать файл клиенту базы данных для выполнения (дополнительная информация о перенаправлении и объединении в конвейеры приводится в приложении А «Основы командной строки»).

За дополнительной информацией об использовании этих команд и об особенностях команды syncdb обращайтесь к главам с примерами приложений в третьей части книги или к официальной документации Django.

Синтаксис запросов

Для обращения к базам данных, созданным с помощью моделей, вам потребуется использовать два отдельных, но похожих друг на друга класса: Manager и QuerySet. Объекты класса Manager всегда подключены к классу модели, поэтому, если вы не определили иначе, каждый ваш класс модели имеет атрибут objects, который является основой для всех запросов к базе данных, касающихся этой модели. Объекты класса Manager – это ворота, ведущие к информации, хранящейся в базе данных. Они имеют тройку методов, позволяющих выполнять типичные запросы:

- all: Возвращает объект класса QuerySet, содержащий все записи в базе данных, принадлежащие текущей модели.

- filter: Возвращает объект класса QuerySet, содержащий записи модели, соответствующие заданному критерию.
- exclude: Выполняет действие, обратное действию метода filter, – отыскивает записи, которые *не* соответствуют заданному критерию.
- get: Получает единственную запись, соответствующую заданному критерию (или возбуждает исключение, если таковых записей не найдено или их больше, чем одна).

Мы немного опережаем события, так как еще не объяснили, что собой представляет класс QuerySet. Объекты класса QuerySet можно рассматривать как списки экземпляров класса модели (или строк/записей в базе данных), хотя они обладают намного более широкими возможностями. Объекты класса Manager представляют собой площадку для создания запросов, а объекты класса QuerySet – место, где происходят основные действия.

Экземпляры класса QuerySet – это многогранные объекты, которые используют преимущества динамической природы языка Python, его гибкость и так называемую утиную типизацию, чтобы предоставить тройку важных и мощных методов. Они являются запросами к базе данных, контейнерами и строительными блоками одновременно.

Класс QuerySet как запрос к базе данных

Как следует из имени класса, объект QuerySet можно рассматривать как запрос к базе данных. Его можно преобразовать в строку с кодом SQL, который выполняется базой данных. Поскольку большинство типичных запросов SQL обычно представляют собой коллекцию логических инструкций и параметров, в том, что объекты QuerySet принимают то же самое на уровне языка Python, есть определенный смысл. Объекты QuerySet принимают именованные аргументы, или параметры, и преобразуют их в соответствующие инструкции на языке SQL. Это наглядно продемонстрировано в примере использования класса Book модели, представленном выше, в этой главе.

```
from myproject.myapp.models import Book

books_about_trees = Book.objects.filter(title__contains="Tree")
```

Допустимые имена аргументов представляют собой комбинацию имен полей модели (таких, как title в предыдущем примере), двух символов подчеркивания в качестве разделителя и необязательных поясняющих слов, таких как contains, gt («greater than» – больше чем), gte («greater than or equal to» – больше либо равно), in (для проверки на принадлежность) и т. д. Каждое из них отображается непосредственно (или почти непосредственно) в операторы и ключевые слова SQL. Полный перечень этих операторов вы найдете в официальной документации.

Вернемся к нашему примеру. Book.objects.filter – это метод класса Manager, как уже говорилось выше, а методы класса Manager всегда воз-

вращают объекты QuerySet. В данном случае мы запросили у менеджера по умолчанию класса Book все записи, в поле title которых содержится слово «Tree», и получили результаты в виде объекта QuerySet, присвоив его переменной. Данный объект QuerySet является представлением запроса SQL, который может выглядеть следующим образом:

```
SELECT * FROM myapp_book WHERE title LIKE "%Tree%";
```

Имеется возможность создавать составные запросы, как, например, для модели Person, определенной выше:

```
from myproject.myapp.models import Person

john_does = Person.objects.filter(last="Doe", first="John")
```

Это соответствует следующему запросу SQL:

```
SELECT * FROM myapp_person WHERE last = "Doe" AND first = "John";
```

Похожие результаты получаются при использовании других упоминавшихся выше методов класса Manager, таких как all:

```
everyone = Person.objects.all()
```

который превращается в незамысловатый запрос SQL:

```
SELECT * FROM myapp_person;
```

Следует заметить, что, как и следовало ожидать, различные дополнительные параметры, имеющие отношение к запросу и определяемые в классе Meta, вложенном в модель, оказывают влияние на создаваемый код SQL – например, параметр ordering превращается в предложение ORDER BY. И, как будет показано ниже, дополнительные методы и составные запросы QuerySet также оказывают влияние на код SQL, который, в конечном счете, передается для выполнения базе данных.

Наконец, если вы изучите SQL и будете понимать сущность различных механизмов запросов (как в терминах результирующего набора данных, так и в терминах времени выполнения), вы окажетесь лучше подготовленными к созданию запросов ORM, более быстрых или более точных, чем те, которые вы сможете создавать без этой подготовки. Кроме того, планируемые и текущие реализации Django упрощают возможность получения открытых объектов QuerySet и настройки соответствующих запросов SQL, предоставляя вам еще большую мощь, чем прежде.

Класс QuerySet как контейнер

Класс QuerySet по своим свойствам напоминает список. Он частично реализует интерфейс класса list и позволяет выполнять итерации по записям (for record in queryset:), обращаться к записям по их индексам (queryset[0]), получать срезы (queryset[:5]) и определять количество полученных записей (len(queryset)). Кроме того, привыкнув рабо-

тать со списками, кортежами или итераторами в языке Python, вы уже будете знать, как использовать QuerySet для доступа к объектам внутри модели. Везде, где только возможно, эти операции выполняются с использованием внутренних механизмов. Например, операции извлечения среза и обращения к элементу по его индексу превращаются в предложения SQL LIMIT и OFFSET.

Иногда может появиться потребность добиться с помощью QuerySet чего-то такого, что нельзя получить с помощью имеющихся возможностей, предоставляемых механизмом ORM платформы Django. В таких случаях вы можете просто передать объект типа QuerySet функции list и получить настоящий список, содержащий полный набор результатов. При этом, хотя иногда бывает необходимо или полезно, например, выполнить сортировку на уровне языка Python, имейте в виду, что такой прием может потребовать значительных объемов памяти или увеличить нагрузку на базу данных, если набор результатов QuerySet содержит значительное число объектов!

Платформа Django стремится обеспечить средствами ORM как можно более широкие возможности, поэтому, если вы действительно подумываете о преобразовании набора результатов в список, то потратьте прежде несколько минут на то, чтобы пролистать эту книгу, просмотреть официальную документацию или архивы рассылок Django. Велика вероятность, что вам удастся решить вашу проблему без переноса всего множества QuerySet в память.

Класс QuerySet как строительный блок

Объекты класса QuerySet выполняют запросы к базе данных, только когда в этом действительно возникает необходимость, например при преобразовании в список или при обращении к ним другими способами, упоминавшимися в предыдущем разделе. Такое поведение является одной из самых сильных сторон объектов QuerySet. Они необязательно должны быть самостоятельными и одноразовыми запросами, но могут использоваться для составления сложных и вложенных запросов. По этой причине класс QuerySet обладает многими методами, которыми обладает и класс Manager, – такими как filter, exclude и многими другими. Точно так же, как и родственные им методы класса Manager, эти методы возвращают объекты класса QuerySet, но ограниченные параметрами родительских объектов QuerySet. Проще всего будет пояснить это на примере.

```
from myproject.myapp.models import Person

doe_family = Person.objects.filter(last="Doe")
john_does = doe_family.filter(first="John")
john_quincy_does = john_does.filter(middle="Quincy")
```

С каждым шагом мы уменьшаем объем получаемых результатов, пока не получаем в конце единственный объект результата или небольшое

их число, – в зависимости от того, сколько записей с информацией о людях с именем John Quincy Doe хранится в базе данных. Поскольку все операции выполняются на языке Python, их можно было бы объединить в одну инструкцию:

```
Person.objects.filter(last="Doe").filter(first="John").filter(middle="Quincy")
```

Конечно, проницательный читатель обратит внимание, что здесь нет ничего такого, что нельзя было бы сделать с помощью единственного вызова `Person.objects.filter`. Однако, можно ли сказать, что мы не получаем аналог прежнего объекта `john_does` типа `QuerySet` при вызове функции или не извлекаем его из структуры данных? В данном случае мы не знаем конкретное содержимое обрабатываемого запроса, но нам это далеко не всегда и нужно.

Представьте, что мы добавили поле `due_date` (срок возврата) в модель `Book` и нам требуется получить список книг с истекшим сроком возврата (то есть те книги, у которых срок возврата истек вчера или еще раньше). При этом мы можем иметь дело с объектом `QuerySet`, содержащим все книги, хранящиеся в библиотеке, или только беллетристику, или книги, которые вернул определенный человек, – то есть `QuerySet` может представлять некоторую произвольную коллекцию книг. От нас требуется взять такую коллекцию и оставить только те книги, которые нас интересуют, а именно, книги с истекшим сроком возврата.

```
from myproject.myapp.models import Book
from datetime import datetime
from somewhere import some_function_returning_a_queryset

book_queryset = some_function_returning_a_queryset()
today = datetime.now()
# __lt превращается в оператор меньше-чем (<) языка SQL
overdue_books = book_queryset.filter(due_date__lt=today)
```

Кроме подобной фильтрации, составные запросы `QuerySet` оказываются необходимы для реализации сложных логических конструкций – например, поиск всех книг, написанных автором с именем Smith и относящихся к разряду научной литературы.

```
nonfiction_smithBook.objects.filter(author__last="Smith").exclude(
                                                        genre="Fiction")
```

Того же результата можно было бы добиться за счет включения в запрос параметра отрицания, такого как `__genre__neg` или подобного ему (такие параметры механизм ORM платформы Django поддерживал в прошлом), однако выделение логики в отдельные методы `QuerySet` делает ее составляющие более обособленными. Кроме того, реализацию проще воспринимать, если она делится на отдельные «шаги».

Сортировка результатов запросов

Следует заметить, что объекты типа QuerySet обладают дополнительными методами, отсутствующими в объектах типа Manager, которые используются только для воздействия на результаты и не создают новых запросов. Наиболее часто используется метод order_by, который переопределяет порядок сортировки по умолчанию. Например, предположим, что предыдущий класс Person по умолчанию выполняет сортировку по фамилии, – тогда мы могли бы получить объект QuerySet, выполняющий сортировку по имени, как показано ниже:

```
from myproject.myapp.models import Person

all_sorted_first = Person.objects.all().order_by('first')
```

В результате мы получили объект QuerySet, ничем не отличающийся от любого другого, но за кулисами, на уровне языка SQL, предложение ORDER BY было изменено в соответствии с нашими потребностями. Это означает, что мы можем продолжать наслаивать синтаксис для получения более сложных запросов, таких как поиск первых пяти человек в списке, отсортированном по имени.

```
all_sorted_first_five = Person.objects.all().order_by('first')[:5]
```

Можно даже выполнить сортировку по отношениям модели, используя синтаксис с двумя символами подчеркивания, который вы уже видели раньше. Представим на минуту, что в нашей модели Person имеется поле ForeignKey, связывающее ее с моделью Address, содержащей, кроме всего прочего, поле state (область, край), и нам необходимо упорядочить список людей сначала по названию области, а затем по фамилии. Сделать это можно было бы следующим способом:

```
sorted_by_state = Person.objects.all().order_by('address__state', 'last')
```

Наконец, порядок сортировки можно изменить на обратный, предваряя идентификационную строку знаком минус, например order_by('-last'). Можно изменить порядок сортировки на обратный для всего объекта QuerySet (если, к примеру, ваш программный код получает объект QuerySet извне и у вас нет возможности напрямую изменить предыдущий вызов order_by) – вызовом метода reverse.

Другие способы изменения запросов

Помимо возможности изменения упорядочения имеется еще несколько методов, которыми обладают только объекты класса QuerySet, – такие как distinct. Этот метод удаляет все повторяющиеся записи в объекте QuerySet, используя для этого инструкцию SELECT DISTINCT на стороне SQL. Метод values принимает список имен полей (включая поля объектов, с которыми установлены отношения) и возвращает объект подкласса ValuesQuerySet класса QuerySet, содержащий только затребованные поля в виде списка словарей вместо обычных объектов класса

модели. Кроме того, у метода values имеется двойник – метод values_list, возвращающий список кортежей. Ниже приводится пример использования методов values и values_list в интерактивной оболочке.

```
>>> from myapp.models import Person
>>> Person.objects.values('first')
[{'first' : u'John'}, {'first': u'Jane'}]
>>> Person.objects.values_list('last')
[(u'Doe',), (u'Doe',)]
```

Еще один полезный метод, который часто остается без внимания, – это метод select_related. Иногда он может помочь в решении типичной проблемы ORM, когда для концептуально простой операции выполняется чрезмерно большое число запросов. Например, при выполнении большого числа циклов по объектам класса Person и последующем выводе информации из связанных с ними объектов Address (продолжая пример из предыдущего раздела) производится один запрос к базе данных для получения списка объектов Person и затем множество запросов для получения каждого объекта Address в отдельности. При таком подходе потребуется выполнить слишком много запросов, если список будет содержать информацию о сотнях или тысячах людей.

Чтобы избежать этого, метод select_related автоматически выполняет соединение таблиц, чтобы «предварительно заполнить» связанные объекты, поэтому фактически выполняется единственный большой запрос – как правило, базы данных лучше справляются с небольшим числом больших запросов, чем с большим числом небольших запросов. Однако обратите внимание, что метод select_related не следует за отношениями, когда определен параметр null=True, поэтому имейте это в виду, когда будете проектировать схему модели для достижения высокой производительности.

И еще несколько заключительных замечаний по использованию метода select_related. С помощью аргумента depth можно контролировать, как «далеко» он будет следовать по цепочке отношений, чтобы, при наличии глубокой иерархии объектов, воспрепятствовать созданию по-настоящему гигантских запросов. Кроме того, при наличии широкой иерархии, с большим количеством связей между объектами имеется возможность выбирать только определенные отношения, передавая имена полей в виде позиционных аргументов.

Например, ниже показано, как можно было бы использовать метод select_related для выполнения простого соединения объектов Person и Address и устранить участие в соединении других полей ForeignKey, которые могут присутствовать в модели Person или Address:

```
Person.objects.all().select_related('address', depth=1)
```

Здесь нет ничего экстраординарного, но он заслуживает внимания; метод select_related и другие полезны, только когда вам требуется получить результат, близкий к тому, что дает механизм запросов по умол-

чанию. Если вам не приходилось прежде работать со средними или крупными веб-сайтами, эти методы не покажутся вам слишком полезными, но они станут совершенно необходимы, когда разработка приложения будет завершена и вас начнет волновать вопрос производительности!

Дополнительную информацию обо всех этих функциях, а также о методах order_by и reverse можно найти в официальной документации к платформе Django.

Именованные аргументы запросов в комбинации с Q и ~Q

Возможности объектов QuerySet могут быть дополнены классом с именем Q, инкапсулирующим именованные аргументы и позволяющим воспроизводить еще более сложные логические конструкции, такие как сочетания логических операций И и ИЛИ, с помощью операторов & и | (которые не следует путать с эквивалентными логическими операторами языка Python and и or или с битовыми операторами & и |). Получающиеся объекты класса Q могут использоваться вместо именованных аргументов в вызовах методов filter и exclude, например:

```
from myproject.myapp.models import Person
from django.db.models import Q

specific_does = Person.objects.filter(last="Doe").exclude(
    Q(first="John") | Q(middle="Quincy")
)
```

Этот пример достаточно искусственный – едва ли вам часто будут попадаться ситуации, когда потребуется выполнить поиск по определенному имени *или* отчеству – его назначение состоит лишь в том, чтобы показать, как используются объекты класса Q.

Подобно объектам QuerySet объекты Q могут конструироваться в процессе выполнения. Когда операторы & или | применяются к объектам Q, они возвращают новые объекты Q, эквивалентные операндам. Например, с помощью цикла можно создавать достаточно большие запросы:

```
first_names = ["John", "Jane", "Jeremy", "Julia"]

first_name_keywords = Q() # Пустая заготовка "запроса" для сборки
for name in first_names:
    first_name_keywords = first_name_keywords | Q(first=name)

specific_does = Person.objects.filter(last="Doe"
).filter(first_name_keywords)
```

Как видите, мы создали короткий цикл for по элементам нашего списка, который «добавляет» имена из списка в получающийся объект Q с помощью оператора |. В действительности этот пример ничем не лучше предыдущего – такая простая задача имеет более оптимальное решение, основанное на применении оператора запроса __in, но мы наде-

6 Зак. 1496

емся, что он наглядно показывает потенциальные возможности объединения объектов Q программным способом.

Примечание

Мы могли бы сократить предыдущий пример на несколько строк, задействовав средства функционального программирования, имеющиеся в языке Python, а именно, генераторы списков, встроенные методы reduce и модуль operator. Модуль operator содержит функциональные аналоги операторов, такие как or_ для оператора | и and_ для оператора &. Три строки, окружающие цикл for, можно было бы переписать, как reduce(or_,[Q(first=name) for name in first_names]). Как всегда, поскольку Django – это «всего лишь Python» такие приемы могут применяться практически к любым аспектам платформы.

Наконец, к объектам Q можно применять унарный оператор ~, инвертирующий содержимое объекта. Хотя для таких нужд чаще применяется метод exclude класса QuerySet, тем не менее, ~Q оказывается лучшим решением, когда логика запроса становится чуть более сложной. Возьмем, например, следующую составную инструкцию, которая извлекает все записи с информацией о людях с фамилией Doe плюс все записи с информацией о людях, имеющих имя John Smith, но исключает записи с информацией о людях, имеющих имя John W. Smith.

```
Person.objects.filter(Q(last="Doe") |
    (Q(last="Smith") & Q(first="John") & ~Q(middle__startswith="W"))
)
```

Применение метода exclude(middle_startswith="W") к такому запросу не даст ожидаемого эффекта, так как он исключит из результирующего набора все записи с информацией о людях с фамилией Doe, отчество которых начинается с символа «W», а это совсем не то, что нам требуется; однако применение конструкции ~Q позволило нам четко выразить свои требования.

Настройка SQL с помощью метода extra

В завершение рассказа о том, чего можно добиться с помощью механизма запросов Django (и как вступление к следующему разделу, где рассказывается о том, что недоступно этому механизму), мы исследуем метод extra класса QuerySet. Этот универсальный метод, используемый для изменения некоторых аспектов простых запросов SQL, которые генерируются объектами QuerySet, принимает четыре именованных аргумента, перечисленные в табл. 4.2. Обратите внимание, что в примерах этого раздела для большей наглядности используются атрибуты, которые не определялись в предыдущих примерах моделей.

В параметре select ожидается словарь идентификаторов, отображаемых в строки SQL, что дает возможность добавлять собственные атрибуты в экземпляры класса модели, основываясь на предложения SELECT запросов SQL. Это удобно, когда необходимо определить простые до-

полнения к информации, извлекаемой из базы данных, и когда эти дополнения необходимы только в ограниченных фрагментах программного кода (в противоположность методам модели, результаты работы которых используются повсеместно). Кроме того, некоторые операции выполняются быстрее в базе данных, чем в программном коде на языке Python, что может использоваться для достижения более высокой производительности приложения.

Таблица 4.2. Некоторые параметры метода extra

Параметр метода extra	Описание
select	Модифицирует части инструкции SELECT
where	Добавляет дополнительные предложения WHERE
tables	Добавляет дополнительные таблицы
params	Выполняет подстановку динамических параметров

Ниже приводится пример использования параметра select для добавления в запрос простой проверки, выполняемой на уровне базы данных, результат которой возвращается в виде дополнительного атрибута:

```
from myproject.myapp.models import Person

# SELECT first, last, age, (age > 18) AS is_adult FROM myapp_person;
the_folks = Person.objects.all().extra(select={'is_adult': "age > 18"})

for person in the_folks:
  if person.is_adult:
    print "%s %s is an adult because they are %d years old." % (person.first,
        person.last, person.age)
```

В параметр where принимается список входных строк, содержащих простые предложения WHERE языка SQL, которые внедряются в запрос SQL как есть (или почти как есть – смотрите описание параметра params ниже). Параметр where лучше всего использовать в ситуациях, когда просто невозможно создать нужный запрос с использованием именованных аргументов, связанных с атрибутами, таких как __gt или __icontains. В следующем примере мы использовали одну и ту же конструкцию SQL для поиска и получения конкатенации строк, задействовав оператор конкатенации ||, в стиле базы данных PostgreSQL:

```
# SELECT first, last, (first||last) AS username FROM myapp_person WHERE
# first||last ILIKE 'jeffreyf%';
matches = Person.objects.all().extra(select={'username': "first||last"},
    where=["first||last ILIKE 'jeffreyf%'"])
```

Параметр tables является, пожалуй, самым простым параметром метода extra. Он позволяет указать имена дополнительных таблиц. Эти имена затем добавляются в предложение FROM запроса, часто используемое в комбинации с инструкцией JOIN. Помните, что по умолчанию

платформа Django присваивает таблицам имена в формате appname_modelname.

Ниже приводится пример использования параметра tables, который несколько отличается своей краткостью от других примеров (и возвращает объект класса Book с дополнительным атрибутом author_last):

```
from myproject.myapp.models import Book

# SELECT * FROM myapp_book, myapp_person WHERE last = author_last
joined = Book.objects.all().extra(tables=["myapp_person"], where=["last =
    author_last"])
```

Наконец мы подошли к аргументу params. Один из «рекомендуемых приемов» выполнения запросов к базе данных на языках программирования высокого уровня заключается в том, чтобы корректным образом экранировать передаваемые значения или использовать динамические параметры. Начинающие программисты часто допускают ошибку, используя простую операцию конкатенации строк для добавления значений переменных в текст запроса SQL, что чревато появлением массы ошибок и проблем с безопасностью.

Вместо этого при использовании метода extra добавляйте именованный аргумент params, который является обычным списком значений, используемых для замещения шаблонных символов %s в строках аргумента where, например:

```
from myproject.myapp.models import Person
from somewhere import unknown_input

# Неправильно: будет "работать", но чревато нападениями типа
# инъекции SQL и связанных с ними проблем.
# Обратите внимание, что символы '%s' замещаются с применением
# обычного механизма вставки строк языка Python.
matches = Person.objects.all().extra(where=["first = '%s'" %
unknown_input()])

# Правильно: кавычки и другие специальные символы будут экранированы
# с учетом особенностей используемой базы данных.
# Обратите внимание, что символы '%s' не замещаются с применением
# обычного механизма вставки строк, а заполняются значениями
# из списка в аргументе 'params'.
matches = Person.objects.all().extra(where=["first = '%s'"],
    params=[unknown_input()])
```

Использование возможностей SQL, не предоставляемых платформой Django

Механизм ORM, реализующий в платформе Django архитектуру модель/запрос, просто не в состоянии покрыть все имеющиеся возможности языка SQL. Немногие ORM могут заявить, что на все 100 процентов способны заменить обычные средства взаимодействия с базой

данных, и платформа Django в этом отношении не является исключением, хотя разработчики не прекращают работу над повышением ее гибкости. Иногда бывает необходимо выйти за рамки ORM – особенно тем, кто обладает богатым опытом работы с реляционными базами данных. Ниже следуют разделы, в которых описывается, как это можно сделать.

Определение схемы и собственный начальный запрос SQL

Помимо стандартных таблиц и столбцов большинством СУРБД предоставляются дополнительные возможности, такие как представления или составные таблицы, триггеры, возможность определения «каскадных» действий, выполняемых при удалении или изменении строк, и даже собственных функций и типов данных на языке SQL. Механизм ORM платформы Django, как и большинство других, не поддерживает большую часть таких возможностей, по крайней мере, на момент написания этих строк, но это не означает, что вы не можете ими пользоваться.

Одной из особенностей, недавно добавленных в механизм определения моделей, является возможность определять собственные файлы с **начальными запросами на языке SQL**; файлы должны иметь расширение `.sql` и находиться в каталоге приложения, в подкаталоге `sql`, например, `myproject/myapp/sql/triggers.sql`. Любые такие файлы автоматически выполняются в базе данных при каждом запуске команд утилиты `manage.py`, имеющих отношение к выполнению сценариев SQL, таких как `reset` или `syncdb`, и включаются в вывод команд `sqlall` и `sqlreset`. У этой особенности в утилите `manage.py` имеется собственная команда – `sqlcustom`, которая (как и другие команды `sql*`) выводит весь обнаруженный код SQL.

Используя начальные запросы на языке SQL, вы можете сохранять команды определения схемы базы данных внутри проекта и можете быть уверены, что они обязательно будут выполнены инструментами Django, когда вы производите построение или перестроение базы данных. Ниже перечислено, что можно получить с помощью этой особенности:

- Представления: Так как представления можно рассматривать как таблицы, доступные только для чтения, вы можете обеспечить их поддержку через создание определений моделей, отражающих их структуру, и затем использовать обычный интерфейс запросов платформы Django для взаимодействия с ними. Обратите внимание, что нужно быть очень внимательными, чтобы не выполнить какие-либо команды утилиты `manage.py`, которые могли бы попытаться записать такие модели в базу данных, в противном случае вы неизбежно столкнетесь с проблемами. Как и с любой другой библиотекой SQL, обеспечивающей доступ к представлениям, попытка *записи* данных в представление приведет к появлению ошибки.

- Триггеры и каскады: Оба механизма прекрасно работают совместно с обычными методами ORM, выполняющими вставку или изменение записей, и могут быть определены с помощью файлов начальных запросов на языке SQL с учетом особенностей используемой базы данных (каскадные ограничения могут быть вручную добавлены в вывод команды `manage.py sqlall`, если они не могут быть созданы).
- Собственные функции и типы данных: Функции и типы данных могут быть объявлены в файлах начальных запросов на языке SQL, но при обращении к ним из ORM необходимо будет использовать метод `QuerySet.extra`.

Оснастки: загрузка и вывод данных

Хотя *по своей сути* эта тема напрямую не связана с языком SQL, мы решили включить этот раздел, чтобы показать еще одну возможность платформы Django, имеющую отношение к работе с базами данных за пределами механизма ORM: **оснастку**. Оснастка, которой мы коротко коснулись в главе 2, – это имя набора данных из базы, сохраняемого в простом файле, но не в обычном формате SQL, а в более универсальном (и часто более удобочитаемом) формате, например XML, YAML или JSON.

Наиболее часто оснастки используются для загрузки начальной информации в базу данных после ее создания или пересоздания, например в «предварительно заполненные» таблицы, которые применяются для категоризации данных, вводимых пользователем, или для заполнения базы данных тестовыми данными, используемыми в процессе разработки приложения. Платформа Django поддерживает такую возможность с помощью средства, напоминающего начальные запросы на языке SQL, описанные выше. Для каждого приложения на платформе Django отыскивается подкаталог `fixtures` и внутри него – файл с именем `initial_data.json` (или `.xml`, `.yaml`, или другим поддерживаемым расширением). Затем, всякий раз, когда выполняются команды создания и пересоздания, такие как `manage.py syncdb`, содержимое этих файлов извлекается с помощью модуля сериализации (дополнительные сведения по этой теме вы найдете в главе 9 «Живой блог») и используется для создания объектов базы данных.

Ниже приводится короткий пример файла оснастки в формате JSON для класса модели `Person`:

```
[
    {
        "pk": "1",
        "model": "myapp.person",
        "fields": {
            "first": "John",
            "middle": "Q",
            "last": "Doe"
        }
```

```
    },
    {
        "pk": "2",
        "model": "myapp.person",
        "fields": {
            "first": "Jane",
            "middle": "N",
            "last": "Doe"
        }
    }
]
```

и вывод команды syncdb, выполняющей импортирование данных в базу:

```
user@example:/opt/code/myproject $ ./manage.py syncdb
Installing json fixture 'initial_data' from '/opt/code/myproject/myapp/
                                            /fixtures'.
Installed 2 object(s) from 1 fixture(s)
```

Помимо загрузки начальных данных оснастки также удобно использовать для вывода информации из базы данных в более «нейтральном» (хотя иногда менее эффективном или специфичном) формате, чем при использовании инструментов базы данных. Например, можно вывести информацию приложения на платформе Django из базы данных PostgreSQL и затем загрузить ее в базу данных MySQL, что довольно сложно выполнить без промежуточного этапа преобразования оснастки. Эту операцию можно выполнить с помощью команд dumpdata и loaddata утилиты manage.py.

При использовании команд dumpdata и loaddata предоставляется большая свобода выбора местоположения и имени оснастки, в отличие от операции загрузки начальных данных. Файлы могут иметь любое имя (при условии, что расширение файла соответствует поддерживаемому формату) и располагаться в каталоге fixtures в любом из каталогов, перечисленных в параметре настройки FIXTURES_DIRS, или в любом другом каталоге, при условии, что командам loaddata и dumpdata будет явно указан полный путь к файлу. Например, мы можем вывести содержимое двух объектов Person, импортированных ранее, как показано ниже (параметр indent используется для повышения удобочитаемости).

```
user@example:/opt/code/myproject $ ./manage.py dumpdata -indent=4 myapp >
➥/tmp/myapp.json
user@example:/opt/code/myproject $ cat /tmp/myapp.json
[
    {
        "pk": 1,
        "model": "testapp.person",
        "fields": {
            "middle": "Q",
            "last": "Doe",
            "first": "John"
```

```
            }
        },
        {
            "pk": 2,
            "model": "testapp.person",
            "fields": {
                "middle": "N",
                "last": "Doe",
                "first": "Jane"
            }
        }
    ]
```

Как видите, оснастки представляют собой очень удобный инструмент для работы с данными в формате, работать с которым проще, чем с форматом SQL.

Собственные запросы SQL

Наконец, важно помнить, что если механизм ORM (включая гибкие возможности, предоставляемые методом `extra`) не отвечает вашим потребностям, всегда имеется возможность выполнить свой собственный запрос SQL с использованием низкоуровневого интерфейса доступа к базе данных. Механизм ORM платформы Django также использует этот интерфейс для работы с базой данных. Какие модули используются при этом, зависит от настроек базы данных, и, как правило, они соответствуют спецификации Python DB-API. Просто импортируйте объект `connection`, определение которого находится в модуле `django.db`, получите с его помощью курсор базы данных и выполните запрос.

```
from django.db import connection

cursor = connection.cursor()
cursor.execute("SELECT first, last FROM myapp_person WHERE last='Doe'")
doe_rows = cursor.fetchall()
for row in doe_rows:
    print "%s %s" % (row[0], row[1])
```

Примечание

За дополнительной информацией о синтаксисе и методах, доступных в этих модулях, обращайтесь к документации Python DB-API, к главе «Database» в книге «Core Python Programming» и/или к документации, описывающей интерфейс доступа к базе данных (*withdjango.com*).

В заключение

В этой главе мы рассмотрели значительный объем базовых сведений, и это здорово, если вы покидаете ее с определенными представлениями, которые могут (и должны) помочь вам приступить к использова-

нию моделей данных. Вы узнали, насколько полезным может быть механизм ORM, как определять модели данных – простые и более сложные, включающие различные отношения, а также познакомились со специальными вложенными классами, используемыми платформой Django для определения метаданных моделей. Кроме того, мы надеемся, что вы убедились в гибкости и широких возможностях класса QuerySet как средства извлечения информации из моделей и поняли, как работать с данными, не используя механизм ORM.

В следующих двух главах – в главе 5 «Адреса URL, механизмы HTTP представления» и в главе 6 – вы узнаете, как использовать модели данных в контексте веб-приложения, включая запросы в логику работы контроллера (представления) и отображая данные в шаблонах.

5

Адреса URL, механизмы HTTP и представления

В предыдущей главе вы узнали, как определять модели данных, формирующие основу большинства веб-приложений. В главе, следующей за этой, будет показано, как отображать эти модели с помощью языка шаблонов и форм платформы Django. Однако сами по себе эти две составляющие веб-платформы могут не очень много – для их работы необходима управляющая логика, которая будет определять, какие данные отобразить, какой шаблон использовать для данного адреса URL и какие действия выполнить для этого адреса.

В этой главе описываются детали реализации в платформе Django архитектуры запрос-ответ протокола HTTP, представленной в главе 3 «Начало», сопровождаемые представлением простых функций Python, формирующих логику контроллера, а также некоторых вспомогательных встроенных функций, используемых для решения типичных задач.

Адреса URL

Механизм связывания адреса URL запроса и ответа с результатами является ключевым для любой платформы разработки веб-приложений. В платформе Django используется чрезвычайно простой, но мощный механизм, позволяющий отображать элементы конфигурации на методы представлений с помощью регулярных выражений, а также связывать эти списки отображений, включая их друг в друга. Такая система отличается простотой в использовании и обладает практически неограниченной гибкостью.

Введение в URLconf

Отображения, упомянутые выше, хранятся в виде файлов с программным кодом на языке Python, которые называются **URLconf**. Эти файлы должны определять объект urlpatterns, получаемый в результате вызова функции patterns платформы Django. При вызове функции patterns передаются следующие аргументы:

- Строка префикса, которая может быть пустой строкой
- Один или более кортежей, каждый из которых содержит строку регулярного выражения, соответствующего единственному адресу URL или множеству адресов; объект функции представления или строку с ее именем; и, возможно, словарь аргументов для этой функции представления

Ниже приводится пример расширенной версии URLconf из приложения блога, созданного нами в главе 2 «Django для нетерпеливых: создание блога», поясняющий вышесказанное:

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

urlpatterns = patterns('myproject.myapp.views',
    (r'^$', 'index'),
    (r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$',
     'archive'),
)
```

Совершенно понятно, что гвоздем программы здесь являются регулярные выражения (впервые представленные в главе 1, «Практическое введение в Python для Django»). Помимо отсутствия ведущих символов слеша (которые опущены ввиду их обязательного наличия в адресах URL), первое, на что следует обратить внимание, это символы регулярных выражений, обозначающие начало и конец строки, – ^ и $, соответственно.

На практике символ ^ используется почти всегда, чтобы устранить неоднозначность соответствия. Адрес URL /foo/bar/ – это не то же самое, что адрес /bar/, однако оба они будут соответствовать регулярному выражению r'bar/'; регулярное выражение r'^bar/' – более точное, и ему будет соответствовать только последний адрес.

Символ $ также используется достаточно часто, по той же самой причине. Он гарантирует, что регулярному выражению будет соответствовать адрес URL до конца, а не его кусок. Однако в элементах URL, предназначенных для подключения других файлов URLconf, символ $ не используется, потому что префикс URL, куда производится включение, является не концом полного адреса, а только его частью.

Примечание

Рассмотрите внимательно первый кортеж в предыдущем примере. Вы увидите в нем выражение, содержащее только r'^$', которое обозначает корневой адрес URL – / – веб-сайта. Как уже говорилось выше, платформа Django отбрасы-

вает ведущий символ слеша, что в результате дает пустую строку, которая ничего не содержит между началом (^) и концом ($). Это регулярное выражение часто используется в проектах Django для определения индексных, или начальных страниц.

Еще один аспект регулярных выражений, о котором следует упомянуть, – это возможность использовать **именованные группы** (подвыражения, заключенные в круглые скобки и начинающиеся с конструкции ?P<идентификатор>) для сохранения частей URL, которые могут изменяться. Эта особенность обеспечивает средство для определения динамических адресов URL. В предыдущем примере мы определили архивный раздел блога, что дает возможность адресовать отдельные записи по дате их создания. Базовая часть адреса URL при обращении к архиву, то есть к записи в блоге, остается постоянной, изменяется только дата. Как будет показано ниже, значения, сохраненные здесь, передаются указанной функции представления, которая в свою очередь может использовать их для обращения к базе данных или как-то еще.

Наконец, после определения регулярного выражения нужно указать функцию, связанную с этим адресом URL, и, в случае необходимости, дополнительные именованные аргументы (в виде словаря). Значение первого аргумента функции patterns, если это непустая строка, используется в качестве префикса для строки с именем функции. Вернемся к нашему примеру в этом разделе. Обратите внимание на строку префикса 'myproject.myapp.views', при добавлении которой получается полный путь к функциям 'index' и 'archive' в модулях, то есть, 'myproject.myapp.views.index' и 'myproject.myapp.views.archive' соответственно.

Замещение кортежей функциями url

Сравнительно недавно в механизме управления адресами платформы Django появилась функция url, призванная заменить кортежи, о которых говорилось выше, сохраняя при этом структуру определения. Она принимает те же три «аргумента» – регулярное выражение, функцию/строку представления и необязательный словарь с аргументами и добавляет еще один необязательный именованный аргумент: name. Аргумент name – это обычная строка, уникальная в множестве адресов URL, которая может использоваться где бы то ни было для ссылки на этот URL.

Перепишем предыдущий пример с использованием функции url.

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

urlpatterns = patterns('myproject.myapp.views',
    url(r'^$', 'index', name='index'),
    url(r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$',
        'archive', name='archive'),
)
```

Как видите, это простая замена прежнего синтаксиса кортежей. Поскольку это в действительности функция, а не кортеж, в ней становится обязательным то, что раньше считалось лишь соглашением. Первые два аргумента являются обязательными и не имеют имени, хотя аргумент-словарь теперь стал необязательным именованным аргументом kwargs. Кроме того, появился новый необязательный именованный аргумент name.

Примечание

Аргументы kwargs и name являются не позиционными, а именованными, чтобы подтвердить тот факт, что оба они являются необязательными. Вы можете не указывать ни один из них, указывать любой из них или оба сразу. С помощью позиционных аргументов (или кортежей) такую свободу выбора организовать значительно сложнее.

Мы представили подход, основанный на использовании функции url, после описания синтаксиса, основанного на кортежах, потому что он более новый и будет продолжать считаться новинкой, даже когда вы будете читать эти строки, и наверняка еще долго будут существовать файлы URLconf, основанные на применении кортежей, а не функции url. Однако мы настоятельно рекомендуем вам использовать функцию url в своем собственном программном коде – мы постарались показать вам хороший пример и использовали эту функцию в оставшейся части книги, потому что она предлагает гораздо более широкие возможности, чем подход, основанный на применении кортежей.

Наконец, дополнительную информацию об аргументе name и его использовании для ссылки на адреса URL из других частей приложения вы найдете в примерах приложений в третьей части, «Приложения Django в примерах».

Использование нескольких объектов patterns

Одна из интересных возможностей, часто используемая разработчиками приложений на платформе Django, заключается в возможности разложения файлов URLconf на несколько вызовов функции patterns в одном файле – по крайней мере, в файлах, имеющих нетривиальное число элементов. Это возможно благодаря тому, что тип значения, возвращаемого функцией patterns, является внутренним типом объектов Django, который позволяет добавлять объекты, как если бы это был список или другой контейнерный тип. Конкатенация нескольких таких объектов выполняется очень просто, и потому возможно и желательно выделять их, основываясь на строке префикса. Ниже приводится полуабстрактный пример, представляющий адреса URL верхнего уровня, связывающие несколько приложений.

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

urlpatterns = patterns('myproject.blog.views',
```

```
    url(r'^$', 'index'),
    url(r'^blog/new/$', 'new_post'),
    url(r'^blog/topics/(?P<topic_name>\w+)/new/$', 'new_post'),
)

urlpatterns += patterns('myproject.guestbook.views',
    url(r'^guestbook/$', 'index'),
    url(r'^guestbook/add/$', 'new_entry'),
)

urlpatterns += patterns('myproject.catalog.views',
    url(r'^catalog/$', 'index'),
)
```

Обратите внимание на оператор += во втором и третьем вызове функции patterns. К концу файла объект urlpatterns будет содержать конгломерат из всех шести адресов URL с отдельными отображениями, соответствующими различным значениям аргумента префикса. Конечно, внимательный читатель обратит внимание, что это разложение еще не окончательное. Некоторые элементы в разделах «blog», «guestbook» и «catalog» определений адресов URL повторяются. Ниже мы покажем, как еще больше можно упростить эти определения включением других файлов URLconf.

Включение других файлов URLconf с помощью функции include

Разложение на составляющие, которое было продемонстрировано в предыдущем разделе, можно продолжить, разбив один файл URLconf на несколько таких файлов. Очень часто можно встретить проекты, состоящие из нескольких приложений, среди которых может быть «базовое» приложение, определяющее индексную страницу или реализующее другие возможности, такие как аутентификация, – обслуживающие весь сайт в целом. При такой организации в файле URLconf базового приложения определяются подразделы, заполняемые другими приложениями и использующие специальную функцию include, которая выполняет перенаправление адресов URL для дальнейшего анализа указанным приложениям, как показано ниже в модификации предыдущего примера.

```
## urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.blog.views',
    url(r'^$', 'index'),
    url(r'^blog/', include('myproject.blog.urls'),
)

urlpatterns += patterns('',
    url(r'^guestbook/', include('myproject.guestbook.urls')),
)
```

```
urlpatterns += patterns('',
    url(r'^catalog/', include('myproject.catalog.urls')),
)

## blog/urls.py

urlpatterns = patterns('myproject.blog.views',
    url(r'^new/$', 'new_post'),
    url(r'^topics/(?P<topic_name>\w+)/new/$', 'new_post'),
)

## guestbook/urls.py

urlpatterns += patterns('myproject.guestbook.views',
    url(r'^$', 'index'),
    url(r'^add/$', 'new_entry'),
)

## catalog/urls.py

urlpatterns += patterns('myproject.catalog.views',
    url(r'^$', 'index'),
)
```

Этот пример получился несколько больше предыдущего, но его преимущество – в применимости к реальным веб-сайтам с десятками адресов URL в каждом разделе. В любом случае, мы прилично сэкономили, избежав ввода повторяющихся строк «blog», «guestbook» и «catalog» в определениях URL. Теперь у нас имеется веб-сайт, состоящий из нескольких приложений, где анализ большинства URL делегируется самим приложениям, за исключением индексной страницы, которая располагается в приложении `blog` (хотя для обслуживания этой страницы можно было бы создать отдельное приложение, скажем, `base` – все в ваших руках).

Возможность включения файлов URLconf может оказаться полезной даже внутри единственного приложения – не существует жестких ограничений, которые определяли бы, когда использовать несколько приложений, а когда единственное, поэтому вполне возможно иметь приложение на платформе Django с сотнями адресов URL. Большинство разработчиков в таких ситуациях сразу же организуют реализацию приложения в виде модулей и используют поддержку включения файлов URLconf. Вообще, внутренняя организация сайта – это ваше дело, а механизмы URLconf обеспечивают максимально возможную гибкость, в которой возможность включения играет не последнюю роль.

Объекты функций и строки с именами функций

Повсюду в этом разделе мы использовали строки, чтобы определять пути к модулям Python, в которых находятся функции представлений, связанные с данными адресами URL. Однако это не единственный возможный способ – платформа Django позволяет также передавать вместо строк вызываемые объекты, как показано ниже:

```
from django.conf.urls.defaults import *
from myproject.myapp import views

urlpatterns = patterns('',  # Префикс больше не требуется
    url(r'^$', views.index),
    url(r'^blog/', include('myproject.blog.urls')),
)
```

Это открывает доступ к более широким возможностям, таким как применение декораторов для обертывания функций универсальных представлений или даже создание собственных вызываемых объектов для реализации более сложных схем делегирования действий различным функциям представлений. Дополнительно о применении декораторов и других приемах, связанных с использованием вызываемых объектов представлений, рассказывается в главе 11 «Передовые приемы программирования в Django».

Примечание

Иногда при использовании вызываемых представлений возникает соблазн применить в файле URLconf инструкцию `from myproject.myapp.views import *`, но это может привести к проблемам при наличии нескольких модулей представлений – представьте себе два отдельных модуля представлений, в каждом из которых определяется собственная функция представления `index`. Поэтому лучше будет импортировать каждый модуль представления по отдельности, как это сделано в предыдущем примере (используя при необходимости конструкцию `from x import y as z`), обеспечивая большую ясность в локальном пространстве имен.

Моделирование HTTP: запросы, ответы и промежуточная обработка

Теперь вы знаете, как определять адреса URL и связывать их с функциями представлений. Теперь настало время подробно рассмотреть среду, окружающую эти функции. Как описывалось в главе 3, платформа Django моделирует протокол HTTP в виде относительно простой схемы запрос-ответ с помощью объектов на языке Python, представляющих запросы и ответы. Наряду с механизмом распределения адресов URL и функциями представлений запросы к вашему веб-приложению проходят следующие этапы:

- Запрос HTTP достигает веб-сервера.
- Веб-сервер передает запрос платформе Django, которая создает объект запроса.
- В соответствии с определениями в файлах URLconf Django отыскивает нужную функцию представления.
- Вызывается функция представления, которой передается объект запроса и все аргументы URL.

- Функция представления создает и возвращает объект ответа.
- Платформа Django преобразует этот объект в формат, понятный веб-серверу.
- Веб-сервер отправляет ответ клиенту.

Сначала мы рассмотрим объекты запросов и ответов, а также их компоненты, после чего перейдем к механизму промежуточной обработки платформы Django, который предоставляет «обработчики» для различных стадий представленного выше процесса. Затем, в следующем большом разделе, мы расскажем все, что вы должны знать о самих функциях представлений.

Объекты запросов

После настройки файлов URLconf вам необходимо определить реакцию приложения на эти адреса URL. Мы вскоре рассмотрим методы представлений, а пока покажем вам структуру объектов, представляющих запросы и ответы HTTP, с которыми имеют дело представления. Все функции представлений принимают параметр `request`, в котором передается объект типа `HttpRequest`, содержащий набор атрибутов, представляющих обычный запрос HTTP, полученный веб-сервером.

Словари GET и POST

Наиболее типичными элементами данных, связанными с запросами HTTP, которые используются разработчиками веб-приложений, являются структуры данных **GET** и **POST**, являющиеся атрибутами объекта `HttpRequest` (как вы могли догадаться, `request.GET` и `request.POST`) и представленные словарями языка Python. Идентичные по своей структуре, они заполняются двумя разными способами, причем важность эти отличий значительно больше, чем можно было себе представить на первый взгляд. Вместе они обеспечивают гибкий способ параметризации веб-запросов.

Примечание

Несмотря на тесную связь со встроенным в язык Python типом данных `dict`, атрибуты `GET` и `POST` класса `HttpRequest` в действительности являются экземплярами класса `QueryDict`, являющегося подклассом класса `dict` и имитирующего поведение этих структур данных в соответствии со спецификацией HTTP CGI. Все пары ключ-значение хранят свои значения в виде списков, даже в случае единственного значения, чтобы корректно обрабатывать ситуации, когда сервер HTTP возвращает несколько значений. Для большего удобства при использовании объектов `QueryDict` в качестве словарей они возвращают единственный элемент. Когда вам потребуется получить несколько значений, вы можете использовать методы объектов `QueryDict`, такие как `getlist`.

Параметры GET передаются как часть строки URL, но технически они не являются частью самого адреса, в том смысле, что они не определяют

другой ресурс (или представление), а только влияют на поведение ресурса, к которому они присоединяются. Например, адрес /userlist/ может указывать на страницу со списком пользователей сообщества веб-сайта. Если разработчик пожелает разделить список на части, чтобы он не был слишком большим, он может указывать номер страницы в виде параметра GET: /userlist/?page=2. Для обработки этого URL будет использоваться то же самое представление, но разработчик сможет проанализировать значение ключа page в словаре GET и вернуть нужную страницу, как показано в следующем абстрактном примере:

```
def userlist(request):
    return paginated_userlist_page(page=request.GET['page'])
```

Заметьте, что при работе с атрибутом request.GET, как и с другими атрибутами объекта запроса, имитирующими поведение словарей, полезно использовать методы словарей, такие как get (смотрите главу 1 «Практическое введение в Python для Django», чтобы освежить свои сведения о словарях), благодаря чему работоспособность приложения не пострадает при попытке обратиться к отсутствующему параметру.

Параметры POST не являются частью URL, они скрыты от пользователя и часто генерируются формами HTML, находящимися внутри веб-страниц. Атрибут action тега FORM определяет, по какому адресу URL будут передаваться данные – когда пользователь выполняет отправку формы, производится вызов URL со словарем POST, содержащим поля формы. Так действуют большинство веб-форм, хотя с технической точки зрения они способны отправлять свои данные и методом GET. (Обычно этого не делается, так как это приводит к созданию длинных и трудночитаемых строк URL и не дает никаких преимуществ.)

Помимо атрибутов GET и POST, объекты обладают еще словарем REQUEST, который объединяет в себе содержимое первых двух атрибутов. Это может быть удобно в ситуациях, когда искомая пара ключ/значение может передаваться представлению любым из двух методов и заранее неизвестно, какой из методов использовался. Однако, следуя принципу «явное лучше неявного» языка Python, большинство опытных программистов приложений на платформе Django не используют эту возможность.

Сеансы и cookies

Следующий за атрибутами GET и POST по частоте использования при работе с объектами запросов – атрибут request.COOKIES – еще один словарь, в котором пары ключ/значение хранят cookies HTTP, поставляемые вместе с запросом. Cookies обеспечивают для веб-страниц возможность сохранения данных в броузере пользователя. Они являются основой большинства систем аутентификации во Всемирной паутине и используются некоторыми коммерческими сайтами для отслеживания истории посещений.

В большинстве случаев cookies используются для обеспечения такой возможности, как сеансы. То есть веб-страница может запросить у броузера некоторое значение, идентифицирующее пользователя (которое сохраняется, когда пользователь выполняет первое подключение к сайту или когда проходит процедуру регистрации), и использовать эту информацию для настройки поведения страницы в соответствии с личными предпочтениями пользователя. Поскольку cookies легко можно изменить на стороне клиента, это делает их небезопасными для хранения каких-нибудь важных данных. Большинство веб-сайтов хранит информацию в объекте сеанса на стороне сервера (обычно в базе данных сайта), а в cookie поставляет только уникальный идентификационный номер сеанса.

Сеансы часто используются, чтобы обеспечить сохранение состояния, так как протокол HTTP по своей природе не имеет такой возможности – каждый цикл запрос/ответ выполняется отдельно и не имеет никакой информации о предыдущих запросах, как и не имеет способа передать информацию последующим запросам. Благодаря сеансам веб-приложения могут обойти этот недостаток, сохраняя элементы данных, такие как сообщение для пользователя об успешном сохранении данных формы, на стороне сервера и передавая их в последующих ответах.

В платформе Django сеансы представлены в виде еще одного атрибута словаря объекта `HttpRequest`: `request.session` (обратите внимание: в имени `session`, в отличие от имен других атрибутов, используются только символы нижнего регистра, потому что сеансы в действительности не являются частью протокола HTTP). Подобно атрибуту `COOKIES`, представленному выше, значение атрибута `session` может извлекаться и записываться с помощью программного кода на языке Python. При первом обращении к этому атрибуту из программного кода его значение извлекается из базы данных на основе cookie сеанса. При записи нового значения в этот атрибут изменения записываются в базу данных, благодаря чему их можно будет получить позднее.

Прочие переменные сервера

Описанные выше атрибуты объекта запроса используются чаще всего, однако запросы содержат массу другой информации, обычно в виде переменных, доступных только для чтения, – одни из них соответствуют требованиям спецификации протокола HTTP, а другие – атрибуты, специфичные и удобные для платформы Django. Ниже перечислены все непосредственные атрибуты объекта запроса:

- `path`: Часть URL после имени домена, например, `/blog/2007/11/04/`; это также строка, которая обрабатывается файлом URLconf.
- `method`: Одна из двух строк, «GET» или «POST», определяет, какой метод HTTP передачи параметров использовался в этом запросе.

- encoding: Строка с определением кодировки символов, которая должна использоваться для декодирования данных, полученных в составе формы.
- FILES: Словарь, содержащий файлы, выгруженные на сервер с помощью поля формы ввода имени файла, каждый из которых представлен еще одним словарем, с парами ключ/значение, содержащими имя файла, тип содержимого и собственно содержимое файла.
- META: Еще один словарь, содержащий некоторые серверные переменные запроса HTTP, не относящиеся к другим аспектам запроса, включая CONTENT_LENGTH, HTTP_REFERER, REMOTE_ADDR, SERVER_NAME и т. д.
- user: Объект, определяющий, прошел ли пользователь процедуру регистрации. Этот атрибут появляется только при активированном механизме аутентификации Django.
- raw_post_data: Необработанная версия данных POST, полученных вместе с запросом. Практически всегда предпочтительнее использовать request.POST, а не request.raw_post_data; этот атрибут предназначен для проверки данных нестандартными методами.

Объекты ответов

К этому моменту вы прочитали об информации, которая передается функции представления, а теперь мы исследуем информацию, возвращаемую функцией, то есть ответ. С нашей точки зрения ответы выглядят проще, чем запросы, – основной объем данных сосредоточен в теле ответа, которое сохраняется в атрибуте content. Обычно это очень длинная строка с кодом разметки HTML, и она настолько важна для объектов HttpResponse, что они предусматривают несколько способов ее записи.

Наиболее типичный метод заключается в создании объекта – функция HttpResponse принимает строку в виде аргумента и сохраняет ее в атрибуте content.

```
response = HttpResponse("<html>This is a tiny Web page!</html>")
```

Этого вполне достаточно, чтобы получить полноценный объект ответа, который уже можно вернуть веб-серверу для пересылки броузеру пользователя. Однако иногда бывает удобно собирать содержимое ответа по частям; для этого объекты HttpResponse реализуют поведение, отчасти напоминающее поведение файлов, в частности – метод write.

```
response = HttpResponse()
response.write("<html>")
response.write("This is a tiny Web page!")
response.write("</html>")
```

Безусловно, из этого следует, что объект HttpResponse можно использовать везде, где ожидается объект файла, например в утилитах записи данных в формате CSV из модуля csv, что придает значительную гибкость процессу создания информации, возвращаемой конечным пользователям.

Еще одна важная особенность объектов ответов заключается в возможности устанавливать заголовки HTTP, используя объект HttpResponse как словарь.

```
response = HttpResponse()
response["Content-Type"] = "text/csv"
response["Content-Length"] = 256
```

Наконец, платформа Django определяет множество подклассов класса HttpResponse, представляющих наиболее часто используемые типы ответов, такие как HttpResponseForbidden (ответ с кодом состояния HTTP 403) и HttpResponseServerError (похожий ответ, но с кодом HTTP 500, означающий внутреннюю ошибку сервера).

Промежуточная обработка

Несмотря на то, что общая схема работы платформы Django выглядит очень просто – принять запрос, найти соответствующую функцию представления, вернуть ответ, – в ней присутствуют промежуточные уровни обработки, расширяющие возможности платформы и увеличивающие ее гибкость. Одним из таких уровней является **уровень промежуточной обработки** – функции на языке Python, вызываемые на разных стадиях описанного выше процесса, которые могут эффективно изменять входные (модифицируя запрос до того, как он достигнет представления) и выходные (модифицируя ответ, созданный представлением) данные *всего* приложения.

Компонент промежуточной обработки в платформе Django – это обычный класс, реализующий определенный интерфейс, а именно – один из множества методов с такими именами, как process_request или process_view. (Наиболее часто используемые методы мы исследуем в следующих подразделах.) Когда имя класса указывается в кортеже MIDDLEWARE_CLASSES в файле settings.py, платформа Django проведет исследование определения класса и будет вызывать его методы в соответствующие моменты времени. Порядок следования классов в кортеже определяет, в каком порядке будут вызываться их методы.

В состав Django входит несколько встроенных классов промежуточной обработки, одну часть которых полезно использовать повсеместно, а другую – только при использовании «вспомогательных» приложений, таких как платформа аутентификации. За дополнительной информацией об этих классах обращайтесь к официальной документации к платформе Django.

Промежуточная обработка запросов

Со стороны входа промежуточная обработка выполняется классом, реализующим метод process_request, как показано в следующем примере:

```
from some_exterior_auth_lib import get_user

class ExteriorAuthMiddleware(object):
    def process_request(self, request):
        token = request.COOKIES.get('auth_token')
        if token is None and not request.path.startswith('/login'):
            return HttpResponseRedirect('/login/')
        request.exterior_user = get_user(token)
```

Обратите внимание на строку, где выполняется присваивание значения атрибуту request.exterior_user, – она иллюстрирует типичное использование возможности промежуточной обработки: добавляет дополнительные атрибуты в объект запроса. В случае, когда выполнение доходит до этой строки, метод process_request неявно возвращает значение None (в языке Python функции всегда возвращают значение None, если в них отсутствует явный вызов инструкции return), после этого платформа Django продолжает промежуточную обработку средствами других классов и в конечном итоге передает его функции представления.

Однако, если проверка покажет, что пользователь еще не зарегистрировался (и даже не пытается зарегистрироваться!), пользователь будет перенаправлен на страницу регистрации. Это иллюстрирует другие возможные варианты поведения промежуточной обработки – в этот момент допускается возвращать объект класса HttpResponse (или одного из его подклассов), который немедленно будет отправлен клиенту, пославшему запрос. Так как в этот момент выполняется промежуточная обработка запроса, все остальные этапы обработки запроса, включая вызов функции представления, пропускаются.

Промежуточная обработка ответов

Как вы уже наверняка догадались, промежуточная обработка ответов применяется к объектам HttpResponse, возвращаемым функцией представления. Классы промежуточной обработки ответов должны реализовать метод process_response, который принимает параметры request и response и возвращает объект класса HttpResponse или одного из его подклассов, с единственным условием – класс промежуточной обработки может изменить содержимое существующего ответа или создать совершенно новый ответ и вернуть его вместо существующего.

Обычно во время промежуточной обработки ответов производится добавление дополнительных заголовков в ответ. Это могут быть заголовки, добавляемые в каждый ответ, такие как заголовки, имеющие отношение к механизму кэширования, или заголовки, зависящие от некоторых условий, например заголовок Content-Language, устанавливаемый

средствами промежуточной обработки в зависимости от текущего языка.

Ниже приводится тривиальный пример, в котором выполняется поиск строки «foo» и замена ее строкой «bar» во всем тексте, который выводится веб-приложением:

```
class TextFilterMiddleware(object):
    def process_response(self, request, response):
        response.content = response.content.replace('foo', 'bar')
```

Мы могли бы привести более практичный пример, отфильтровывающий нецензурные выражения (что могло бы с успехом использоваться в реализации веб-сайта сообщества, например), но эту книгу могут читать все члены семьи!

Представления/управляющая логика

Представления (они же контроллеры) формируют ядро любого веб-приложения на платформе Django, где сосредоточена практически вся управляющая логика. Определяя и используя модели, мы становимся администраторами баз данных; создавая шаблоны – дизайнерами, а создавая представления, мы становимся настоящими *программистами*.

Хотя представления без труда могут составить большую часть программного кода, тем не менее, слой программного кода платформы Django, окружающий представления, оказывается удивительно тонким. Представления содержат управляющую логику приложения, и потому эта часть веб-приложения меньше всего нуждается в связующем программном коде и больше – в реализации действий. В то же время встроенные *универсальные представления* являются одним из наиболее привлекательных средств экономии времени, имеющихся в веб-платформах, таких как Django, и мы познакомим вас с этими представлениями и методами их использования как отдельно, так и в тандеме с вашими собственными представлениями.

Просто функции на языке Python

В своей основе представления Django являются функциями на языке Python, самыми простыми и обычными функциями. Единственное требование, предъявляемое к ним, заключается в том, что они должны принимать объект `HttpRequest` и возвращать объект `HttpResponse`, которые были описаны выше. Кроме того, выше упоминалось, что в регулярных выражениях, в файлах URLconf, можно использовать именованные сохраняющие группы. В комбинации с необязательным параметром-словарем они могут использоваться для организации передачи дополнительных аргументов в функцию представления, как показано в следующем примере (который представляет собой немного измененную версию примера, приводившегося ранее):

```
urlpatterns = patterns('myproject.myapp.views',
    url(r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$',
        'archive',
        {'show_private': True}),
)
```

В комбинации с объектом HttpRequest сигнатура функции представления archive, на которую ссылается предыдущее определение URL, могла бы иметь такой вид:

```
from django.http import HttpResponse

def archive(request, year, month, day, show_private):
    return HttpResponse()
```

Так как функция возвращает объект класса HttpResponse некоторого вида, то внутренняя ее реализация для нас сейчас не имеет большого значения – все, что мы сейчас видим, это, по сути, API. Как и в случае с любым другим API, вы можете использовать предварительно созданную реализацию или написать свою функцию с самого начала. Мы рассмотрим эти варианты именно в таком порядке.

Универсальные представления

Вероятно, одним из самых рекламируемых аспектов платформы Django и веб-платформ вообще является возможность использовать предопределенный программный код для выполнения так называемых операций CRUD, составляющих большую часть средних веб-приложений. Аббревиатура CRUD происходит от Create, Read (или Retrieve), Update и Delete (создание, чтение (или получение), изменение и удаление) – наиболее типичных действий, выполняемых приложениями, базирующимися на использовании базы данных. Требуется вывести список элементов или страницу с подробными сведениями об одном объекте? Это операция извлечения. Требуется вывести форму с редактируемыми элементами и записать изменения в базу данных? Это операция изменения или создания, в зависимости от приложения и рассматриваемой формы. Операция удаления не требует пояснений.

Решение всех этих и подобных им задач предусмотрено множеством **универсальных представлений**, входящих в состав Django. Как было показано выше, все они являются обычными функциями на языке Python, хотя и сильно абстрагированными и параметризуемыми, для достижения максимальной гибкости в рамках отведенной им роли. Поскольку они реализуют всю необходимую управляющую логику, пользователям платформы достаточно лишь сослаться на них в своих файлах URLconf, передать необходимые параметры и убедиться в наличии шаблона, который будет заполняться и возвращаться представлением.

Например, универсальное представление object_detail предназначено для отображения единственного объекта и принимает свои параметры

как из регулярного выражения URL, так и из словаря, как показано ниже:

```
from django.views.generic.list_detail import object_detail
from django.conf.urls.defaults import *
from myproject.myapp.models import Person

urlpatterns = patterns('',
    url(r'^people/(?P<object_id>\d+)/$', object_detail, {
        'queryset': Person.objects.all()
    })
)
```

В предыдущем примере мы определили регулярное выражение, которому соответствуют такие адреса URL, как /people/25/, где 25 – это идентификационный номер записи в базе данных с информацией о человеке, которую требуется отобразить. Универсальному представлению object_detail требуется передать аргумент object_id и объект класса QuerySet, который может быть отфильтрован в соответствии с указанным идентификационным номером. В данном случае значение аргумента object_id извлекается из URL, а значение аргумента queryset передается через словарь аргументов.

Универсальные представления часто имеют дополнительные параметры. Некоторые из них являются характерными для данного представления, а другие носят глобальный характер – например, аргумент template_name, дающий пользователю возможность переопределять расположение шаблона по умолчанию, или словарь extra_context, позволяющий передавать в контекст шаблона дополнительную информацию. (Подробнее о шаблонах и контекстах рассказывается в главе 6 «Шаблоны и обработка форм».) Полный перечень универсальных представлений и их аргументов вы найдете в официальной документации к платформе Django, а ниже рассматриваются лишь некоторые из наиболее часто используемых. Обратите внимание, что универсальные представления организованы в виде двухуровневой иерархии модулей.

- simple.direct_to_template: Полезно при использовании шаблонов с динамическим содержимым (в противоположность «плоским страницам», содержащим только статическую разметку HTML, – смотрите главу 8 «Система управления содержимым»), не требующих реализации какой-то специфической управляющей логики, например для организации индексных страниц или для вывода плоских/смешанных списков.
- list_detail.object_list и list_detail.object_detail: Эти два представления реализуют возможность вывода информации, доступной только для чтения, необходимую в большинстве веб-приложений, и используются, пожалуй, наиболее часто, так как простое отображение информации обычно не требует сложной логики. Однако, если вам необходимо выполнить некоторые действия по подготовке

содержимого шаблона, возможно, лучше будет реализовать собственное представление.

- `create_update.create_object` и `create_update.update_object`: Удобны для простого создания или изменения объекта, когда все, что вам требуется, – это простая проверка содержимого формы, определяемая вашей формой или моделью (смотрите главы 6 и 4 соответственно), и когда не требуется использовать какую-либо другую управляющую логику.
- `date_based.*`: Несколько универсальных представлений, основанных на датах, которые подчеркивают происхождение Django как платформы, ориентированной на публикацию. Они чрезвычайно удобны при работе с любыми типами данных, основанными на датах. В их число входят представления индексных страниц и страниц с подробной информацией по датам, плюс страницы подсписков с детализацией от года до одного дня.

Универсальные представления – это и благо, и проклятье. Преимущества очевидны – они экономят массу времени и могут использоваться,

Передача полных объектов QuerySet в универсальные представления

На первый взгляд кажется неэффективным передавать результат вызова метода `Person.object.all()`, потому что возвращаемый им объект `QuerySet` может оказаться огромным списком всех объектов `Person`! Однако не забывайте, что говорилось в главе 4 «Определение и использование моделей» – объекты `QuerySet` обычно могут быть отфильтрованы с помощью метода `filter` и/или `exclude` прежде чем они превратятся в фактические запросы к базе данных. Благодаря этой особенности можно быть уверенным, что универсальное представление `object_detail` отфильтрует все не относящиеся к делу объекты и в результате будет получен список вполне приемлемого размера.

Кроме того, принимая полноценный объект `QuerySet` вместо, скажем, класса модели (который предоставляет еще один способ отыскать требуемый объект), платформа Django предоставляет нам возможность выполнить необходимую фильтрацию самостоятельно. Например, в представление `object_detail` страницы с описанием подробностей только о служащих, можно было бы передать не `Person.object.all()`, а `Person.objects.filter(is_employee=True)`.

Как всегда, разработчики Django стремятся принимать такие решения, которые обеспечат вам большую гибкость, даже если получаемые преимущества не видны на первый взгляд.

когда основная работа может быть выполнена с помощью простых или умеренно сложных представлений. Их практичность может быть расширена за счет обертывания их собственными представлениями, как будет продемонстрировано ниже. Однако иногда недостаток особенностей может осложнить их использование – в этом случае вам *следует* написать собственное представление с самого начала, даже если универсальное представление, наиболее полно отвечающее вашим представлениям, способно удовлетворить ваши потребности на 90 процентов. Понимание, когда следует признать себя побежденным и пойти своим путем, – это ценный навык, который, как и многие другие аспекты разработки программного обеспечения, приходит только с опытом.

Полууниверсальные представления

Бывают ситуации, когда возможностей универсальных представлений, вызываемых непосредственно из файла URLconf, недостаточно. Часто в этих случаях приходится писать собственные функции представлений, но не менее часто бывает возможным задействовать в них универсальные представления для выполнения черновой работы в зависимости от требуемой логики выполнения.

Исходя из нашего опыта, наиболее часто потребность в таких «полууниверсальных» представлениях возникала из необходимости обойти некоторые ограничения, свойственные файлам URLconf. Например, у вас нет возможности выполнить обработку сохраненных параметров URL, пока не будет проанализировано регулярное выражение. Это ограничение обусловлено принципом действия URLconf, но его легко можно обойти. Взгляните на следующий фрагмент, где представлены отрывки из файлов URLconf и представления:

```
## urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.myapp.views',
    url(r'^people/by_lastname/(?P<last_name>\w+)/$', 'last_name_search'),
)

## views.py

from django.views.generic.list_detail import object_list
from myproject.myapp.models import Person

def last_name_search(request, last_name):
    return object_list(request,
        queryset=Person.objects.filter(last__istartswith=last_name)
    )
```

Как видите, хотя ваша функция и принимает аргумент `last_name`, определяемый как именованная группа в регулярном выражении URL, тем не менее, 99 процентов фактической работы выполняется универсальным представлением. Такое возможно благодаря тому, что уни-

версальные представления – это обычные функции на языке Python и как обычные функции они могут импортироваться и вызываться. Очень легко попасть в ловушку, если думать о платформе, как об отдельном языке программирования, но ранее мы уже подчеркивали, что все это написано на языке Python и такого рода приемы наглядно показывают это.

Собственные представления

Наконец, как уже отмечалось выше, иногда возникают ситуации, когда универсальные представления вообще неприменимы, что возвращает нас к началу этого раздела – функция представления, соответствующая простому API, пустая, как лист бумаги, ждет вас, программиста, чтобы вы ее заполнили. Мы поделимся с вами парой наблюдений из нашего опыта и покажем несколько удобных сокращений, предлагаемых платформой. Однако в этой области только ваши знания и ваш опыт будут определять, что делать дальше.

Сокращения, предоставляемые платформой

Как уже говорилось выше, как только вы попали в царство собственных представлений, платформа Django оставляет вас в покое. Однако она предоставляет несколько сокращений, большая часть которых сосредоточена в модуле `django.shortcuts`.

- `render_to_response`: Функция, которая замещает двух- или трех-шаговый процесс, включающий в себя создание объекта `Context`, отображение его в объект `Template` и возвращение объекта `HttpResponse`, содержащего результат. Она принимает имя шаблона, необязательный контекст (обычно объект `Context` или словарь) и/или тип MIME и возвращает объект `HttpResponse`. Порядок отображения объекта `Template` описывается в главе 6.
- `Http404`: Подкласс класса `Exception`, который фактически возвращает код ошибки HTTP 404 и отображает шаблон верхнего уровня `404.html` (если вы не переопределили поведение по умолчанию в файле `settings.py`). Чтобы привести его в действие, достаточно просто возбудить это исключение, как любое другое, с помощью инструкции `raise`. Идея состоит в том, что состояние 404 рассматривается, как полноценная ошибка, такая же, как ошибка, которая возникает при попытке сложить строку и число. Определение класса находится в модуле `django.http`.
- `get_object_or_404` и `get_list_or_404`: Эти две функции просто пытаются получить объект или список и в случае неудачи возбуждают исключение `Http404`. Они принимают аргумент `klass`, в котором можно передать класс модели, объект типа `Manager` или `QuerySet`, и некоторые аргументы запроса к базе данных, какие обычно передаются объектам `Manager` и `QuerySet`, и пытаются вернуть требуемый объект или список.

Ниже приводятся два примера использования представленных выше сокращений: в первом используется исключение Http404, а во втором демонстрируется, как то же самое можно выполнить с помощью сокращения get_object_or_404 — с практической точки зрения эти две функции обладают идентичным поведением. Пусть вас пока не волнует путь к шаблону — об этом более подробно будет рассказываться в главе 6.

Ниже приводится способ возбуждения исключения Http404 вручную:

```
from django.shortcuts import render_to_response
from django.http import Http404
from myproject.myapp.models import Person

def person_detail(request, id):
    try:
        person = Person.objects.get(pk=id)
    except Person.DoesNotExist:
        raise Http404

    return render_to_response("person/detail.html", {"person": person})
```

А далее следует пример использования функции get_object_or_404, с помощью которой можно упростить реализацию предыдущего метода

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

def person_detail(request, id):
    person = get_object_or_404(Person, pk=id)

    return render_to_response("person/detail.html", {"person": person})
```

Другие наблюдения

Приведем, возможно, не самое важное замечание — многие разработчики веб-приложений на платформе Django применяют соглашение об аргументах «args/kwargs» при определении собственных функций представлений. Как было показано в главе 1, функции могут использовать конструкции *args и **kwargs, чтобы получить возможность принимать любые позиционные и именованные аргументы. Это обоюдоострое оружие (точные сигнатуры функций часто являются превосходным источником информации, но при таком подходе эта информация теряется), но нередко весьма полезный прием, повышающий гибкость и ускоряющий загрузку. Вам не придется больше все время возвращаться к файлу URLconf, чтобы вспомнить, какие имена были присвоены сохраняющим группам в регулярном выражении или именованным аргументам, — для этого достаточно лишь объявить свою функцию, как показано ниже:

```
def myview(*args, **kwargs):
    # Здесь можно ссылаться на аргументы, как args[0] или kwargs['object_id']
```

и ссылаться на элементы kwargs["идентификатор"] по мере необходимости. Спустя некоторое время этот прием войдет в привычку и многие вещи станут проще, особенно когда будет возникать потребность передать в свою функцию аргументы, предназначенные для другой функции, например при реализации «полууниверсальных» представлений, упоминавшихся выше.

В заключение

Мы прошли уже больше половины пути в исследовании основ базовых компонентов платформы Django. В дополнение к моделям, рассматривавшимся в главе 4, в этой главе вы познакомились с механизмом анализа адресов URL и остальными механизмами платформы Django, обеспечивающими взаимодействия запрос-ответ по протоколу HTTP, включая использование средств промежуточной обработки. Вы также увидели, как составлять простые функции представлений платформы Django, и узнали, как подключать и использовать универсальные представления.

Следующая и последняя глава в этой части книги, глава 6, описывает третью основную проблему, которая связана с отображением веб-страниц с помощью шаблонов и управлением вводом пользователя с помощью форм и механизмов проверки ввода. Далее начинается третья часть книги, где в четырех примерах приложений будет продемонстрировано, как использовать эти концепции.

6

Шаблоны и обработка форм

Теперь, когда вы узнали, как определять модели Django и управляющую логику, пришло время рассмотреть последнюю проблему: как организовать отображение информации и ввод данных пользователем. Начнем с обзора языка шаблонов платформы Django и системы отображения, а во второй части главы перейдем к формам и их обработке.

Шаблоны

Как упоминалось в предыдущих главах, шаблоны – это самостоятельные текстовые файлы, содержащие статическое содержимое (например, код HTML) и динамическую разметку, определяющую логику управления, циклы и отображение данных. Решение о том, какой шаблон использовать и какие данные передавать для отображения, принимается функцией отображения (где отображение выполняется явно или с помощью вспомогательной функции `render_to_response`) или определяется аргументами (такими как аргумент `template_name` в универсальных представлениях).

Язык шаблонов в платформе Django предназначен для использования дизайнерами интерфейса, которые необязательно должны быть программистами. Вследствие этого, а также из-за стремления отделить логику от отображения, язык шаблонов не имеет никакого отношения к языку Python. Однако расширяемая система тегов и фильтров (подробности рассматриваются ниже) позволяет программистам приложений на платформе Django расширять набор конструкций управляющей логики, доступных в языке шаблонов.

Наконец, следует заметить, что обычно система шаблонов используется для создания разметки HTML (все-таки речь идет о Всемирной паутине), но это совершенно необязательно, и она с таким же успехом

может использоваться для создания файлов журналов, содержимого писем электронной почты, файлов CSV и любого другого содержимого в каком-либо текстовом формате. Помня об этом, вы сможете более полно использовать возможности шаблонов платформы Django.

Понимание контекста

Шаблоны, будучи динамическими текстовыми документами, были бы бесполезны, если бы не получали для отображения динамическую информацию. Информация, которая передается шаблону для отображения, в терминах Django называется **контекстом**. Контекст шаблона – это словарь класса `Context`, в котором пары ключ-значение содержат информацию для отображения.

Как коротко отмечалось в главе 2 «Django для нетерпеливых: создание блога», и в главе 5 «Адреса URL, механизмы HTTP и представления», для отображения любого шаблона необходимо предоставить его контекст. Иногда контекст может заполняться без вашего участия, например универсальными представлениями, и вы просто дополняете его с помощью аргумента `extra_context`. Иногда, как в случае с собственными представлениями, вам придется определять контекст самостоятельно при передаче методу `render` или вспомогательной функции `render_to_response`. С технической точки зрения вполне возможно отобразить шаблон с пустым контекстом, но в таких ситуациях лучше использовать вспомогательное приложение `flatpages` – шаблоны без контекста не обладают большой динамичностью.

Другой способ внесения данных в контекст шаблона заключается в использовании **процессоров контекста** – механизма платформы, напоминающего механизм промежуточной обработки, позволяющего определять различные функции, которые добавляют пары ключ-значение ко *всем* контекстам непосредственно перед отображением шаблона. Благодаря этой возможности некоторые механизмы, такие как платформа аутентификации, оказываются в состоянии гарантировать отображение определенных элементов данных, общих для всего сайта. Ниже приводится короткий пример процессора контекста.

```
def breadcrumb_processor(request):
    return {
        'breadcrumbs': request.path.split('/')
    }
```

Вероятно, не слишком полезный пример, так как навигационные цепочки редко используются на практике, но он демонстрирует простоту использования процессоров контекста. Вы можете хранить функции процессоров контекста, где вам заблагорассудится, но лучше определить для них более или менее стандартное место, например файл `context_processors.py` в корневом каталоге проекта или приложения.

Процессоры контекста, так же как и средства промежуточной обработки, активируются в файле settings.py, для чего необходимо добавить ссылки на них, с использованием синтаксиса модулей языка Python, в кортеж TEMPLATE_CONTEXT_PROCESSORS. Еще одна черта сходства со средствами промежуточной обработки заключается в том, что порядок их следования имеет значение – процессоры контекста применяются в том порядке, в каком они перечислены в кортеже.

Синтаксис языка шаблонов

Синтаксис языка шаблонов платформы Django напоминает синтаксис языков шаблонов, не использующих формат XML, таких как Smarty или Cheetah, в том смысле, что он не стремится соответствовать требованиям XHTML и использует специальные символы для отделения переменных шаблона и команд управляющей логики от статического содержимого (роль которого обычно играет разметка HTML). Как и многое другое в платформе Django, язык шаблонов слабо связан с остальной частью платформы, благодаря чему имеется возможность в случае необходимости использовать другую библиотеку шаблонов.

Как и в большинстве других языков шаблонов, существуют простые команды, такие как команда вывода значения контекстной переменной, и блочные – обычно это логические команды, такие как «if» или «for». В языке шаблонов Django используются два соглашения, причем оба связаны с использованием фигурных скобок – вывод значений переменных осуществляется с применением двойных фигурных скобок ({{ переменная }}), а все остальное оформляется в виде тегов ({% команда %}). Ниже приводится простой пример, который отображает содержимое словаря контекста { "title_text": "My Webpage", "object_list": ["One", "Two", "Three"] }.

```
<html>
    <head>
        <title>{{ title_text }}</title>
    </head>
    <body>
        <ul>
        {% for item in object_list %}
            <li>{{ item }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

Следует отметить, что при выводе переменных контекста в шаблоне производится неявный вызов функции unicode, вследствие чего объекты и другие переменные, не являющиеся строками, преобразуются в строки Юникода. Будьте внимательны: если попытаться вывести объект, в котором отсутствует метод __unicode__, он не появится в шаблоне. Это обусловлено тем, что по умолчанию в языке Python формат

представления объектов напоминает теги HTML, то есть текст заключается в символы < и >.

```
>>> print object()
<object object at 0x40448>
```

Это типичная ошибка, с ней иногда сталкиваются даже опытные разработчики Django, поэтому, если при попытке отобразить какой-либо объект он не выводится, то в первую очередь убедитесь, что это тот объект, который вам нужен, и он имеет именно то строковое представление, которое вы ожидаете!

Как видите, несмотря на то, что синтаксис шаблонов платформы Django семантически не является корректной разметкой HTML, тем не менее, применение фигурных скобок упрощает визуальное выделение инструкций вывода и команд в статическом содержимом. Кроме того, команда разработчиков Django предусматривала возможность использования языка шаблонов для вывода документов в форматах, отличных от HTML, поэтому они посчитали, что не имеет смысла создавать систему шаблонов, сконцентрированную на выводе XML.

Фильтры шаблонов

Простая операция вывода значений переменных не отличается особой гибкостью, хотя она составляет основу для создания динамических шаблонов. Система шаблонов позволяет выполнять трансформацию значений переменных контекста посредством механизма под названием **фильтры**, который напоминает конвейеры в UNIX – смотрите приложение А «Основы командной строки», если вы еще не знакомы с конвейерами. В определениях фильтров даже используется тот же самый синтаксис, что и в конвейерах UNIX, – символ |. А благодаря тому, что фильтры всегда принимают и возвращают единственную текстовую строку, они могут объединяться в цепочки. Как будет показано в разделе «Расширение шаблонов» главы 11 «Передовые приемы программирования в Django», фильтры являются обычными функциями на языке Python.

В составе Django имеется большое число разнообразных и полезных фильтров, позволяющих решать наиболее общие задачи веб-разработки и обработки текста, такие как экранирование символов слеша, перевод символов в верхний регистр, форматирование дат, получение длин списков и кортежей, конкатенация строк и т. д. Ниже приводится пример использования фильтров для приведения символов строк в списке к нижнему регистру.

```
<ul>
{% for string in string_list %}
    <li>{{ string|lower }}</li>
{% endfor %}
</ul>
```

Большинство фильтров принимают на входе единственную строку, но некоторые фильтры, такие как yesno, принимают произвольные (обычно логические) значения с дополнительным аргументом, определяющим их поведение, и выводят строки, удобные для восприятия человеком.

```
<table>
    <tr>
        <th>Name</th>
        <th>Available?</th>
    </tr>
    {% for person in person_list %}
    <tr>
        <td>{{ person.name }}</td>
        <td>{{ person.is_available|yesno:"Yes,No" }}</td>
    </tr>
    {% endfor %}
</table>
```

Теги

Как вы могли заметить в предыдущих примерах, возможность фильтрации и вывода значений переменных имеет большое значение, но истинная мощь языка шаблонов заключена в **тегах**. До сих пор мы видели, как теги помогают организовать цикл – для обхода списков строк или объектов, но они могут обеспечивать реализацию управляющей логики (`{% if %}`, `{% ifequal %}`), включение/наследование шаблона (`{% block %}`, `{% include %}` и `{% extends %}`, как будет показано в следующем разделе) и решать многие другие задачи.

С технической точки зрения теги могут иметь любую форму и принимать входные значения, следующие за именем тега, в любом виде (подробнее об этом рассказывается в разделе «Расширение шаблонов», в главе 11). Однако встроенные теги и большинство тегов, определяемых пользователем, следуют определенным соглашениям, принимая входные аргументы в виде списка значений, разделенных пробелами. В большинстве случаев аргументами тегов могут быть переменные контекста и во многих ситуациях могут также использоваться фильтры. Например, в следующем примере показано, как проверить длину списка перед выполнением итераций по его содержимому.

```
{% ifequal object_list|length 10 %}
    <ul>
    {% for item in object_list %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endifequal %}
```

Точно так же мы могли бы использовать фильтр `length_is`, который принимает список и аргумент и возвращает логическое значение.

```
{% if object_list|length_is:10 %}
    <ul>
    {% for item in object_list %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endif %}
```

Этот пример показывает, на наш взгляд, что встроенные фильтры и библиотека тегов платформы Django обладают значительной гибкостью. Есть смысл хорошо ориентироваться в том, что доступно (исчерпывающий перечень приводится в документации Django), чтобы впоследствии не изобретать повторно колесо.

В заключение о тегах следует заметить, что блочные теги, такие как `{% if %}` и `{% for %}`, способны изменять свой локальный контекст, что часто бывает удобно. Например, тег `{% for %}` предоставляет локальную контекстную переменную `{{ forloop }}`, имеющую разнообразные атрибуты, которые позволяют выполнять различные действия в зависимости от используемых атрибутов и порядкового номера итерации. Например, можно выполнять действия в начале или в конце итераций (логические атрибуты `{{ forloop.first }}` или `{{ forloop.last }}` указывают, выполняется ли первая или последняя итерация, соответственно) или в зависимости от порядкового номера итерации (который определяется с помощью атрибутов `{{ forloop.counter }}` и `{{ forloop.counter0 }}`, начинающих отсчет итераций с 1 или с 0 соответственно). Дополнительную информацию о таких переменных и примеры их использования вы найдете в документации к платформе Django.

Блоки и расширение

Существует интересная группа тегов, дающих возможность выйти за рамки текущего шаблона и обеспечить взаимодействие с другими файлами шаблонов, позволяя создавать составные шаблоны и многократно использовать программный код с помощью двух основных методов: наследования и включения. Сначала мы рассмотрим метод наследования, как наиболее естественный способ логической организации шаблонов. Метод включения, хотя и полезный, может привести к появлению «винегрета» из шаблонов, что существенно осложняет отладку и разработку.

Наследование шаблонов реализуется с помощью тегов `{% extends %}` и `{% block %}`. Тег `{% extends %}` должен находиться в начале шаблона. Он указывает механизму отображения, что этот шаблон наследует шаблон более высокого уровня. Например, можно создать шаблон верхнего уровня, определяющий верхний/нижний колонтитулы и средства глобальной навигации; затем промежуточные шаблоны для каждого из подразделов, дополняющие шаблон верхнего уровня (например, добавлением навигационного меню второго уровня); и, наконец, шаблоны нижнего уровня для каждой отдельной страницы сайта, дополняю-

щие промежуточные шаблоны и включающие фактическое содержимое страниц.

Тег {% block %} – это тег блока. Он используется для определения разделов шаблона, которые будут заполняться шаблонами, расширяющими этот шаблон. Обычно блоки используются в дочерних шаблонах, но это совершено необязательно. Блоки могут игнорироваться (в этом случае будет отображаться содержимое, определяемое родительским шаблоном) или заполняться шаблонами более низкого уровня. Ниже приводится упрощенный пример использования трехуровневой схемы организации веб-сайта, упомянутой выше, в котором имеются следующие адреса URL: /, /section1/, /section2/, /section1/page1/ и /section1/page2/.

Пока мы не будем рассматривать индексные страницы корневого раздела сайта и его разделов и сконцентрируемся на страницах нижнего уровня. Как показано ниже, файл base.html представляет верхний уровень структуры, шаблоны разделов добавляют заголовок страницы (тем самым показывая пользователю, в каком разделе сайта он находится) и шаблоны страниц обеспечивают отображение простого содержимого.

base.html:

```
<html>
    <head>
        <title>{% block title %}My Web site{% endblock %}</title>
    </head>
    <body>
        <div id="header">
            <a href="/section1/">Section 1</a>
            <a href="/section2/">Section 2</a>
        </div>
        <div id="content">
            {% block content %}{% endblock %}
        </div>
        <div id="footer">
            <a href="/about/">About The Site</a>
        </div>
    </body>
</html>
```

section1.html:

```
{% extends "base.html" %}

{% block title %}Section 1{% endblock %}
```

section2.html:

```
{% extends "base.html" %}

{% block title %}Section 2{% endblock %}
```

page1.html:

```
{% extends "section1.html" %}

{% block content %}This is Page 1.{% endblock %}

page2.html:

{% extends "section1.html" %}

{% block content %}<p>This is Page 2.</p>{% endblock %}
```

При использовании шаблонов из предыдущего примера при посещении адреса /section1/page2 броузер пользователя увидит следующее:

```
<html>
    <head>
        <title>Section 2</title>
    </head>
    <body>
        <div id="header">
            <a href="/section1/">Section 1</a>
            <a href="/section2/">Section 2</a>
        </div>
        <div id="content">
            <p>This is Page 2.</p>
        </div>
        <div id="footer">
            <a href="/about/">About The Site</a>
        </div>
    </body>
</html>
```

Вся прелесть наследования шаблонов состоит в том, что при этом легко просматривается иерархия шаблонов и на любой странице видно, какому шаблону и какой фрагмент разметки HTML принадлежит. Кроме того, в сравнении с приемом, основанным на включении, когда приходится подключать определения заголовков, нижних колонтитулов, боковых панелей и т. д. к каждой странице, наследование позволяет существенно сэкономить на вводе с клавиатуры.

Включение других шаблонов

Наследование шаблонов имеет множество достоинств, но прием включения также находит применение. Иногда бывает необходимо повторно использовать фрагменты разметки HTML или другого текста, когда такое использование плохо укладывается в схему наследования, как в случае, например, нумерации страниц. Возможность включения в платформе Django поддерживается тегом `{% include %}`, который ведет себя именно так, как можно было бы ожидать, – принимает имя файла шаблона и замещает себя содержимым этого файла. Включаемые файлы сами по себе могут быть полноценными шаблонами, а их содержимое будет анализироваться с учетом контекста включающего шаблона.

В дополнение к тегу {% include %} Django предоставляет тег {% ssi %} (где имя тега ssi происходит от используемой в сервере Apache аббревиатуры SSI, или Server Side Includes – включения на стороне сервера). Теги {% include %} и {% extends %} могут ссылаться только на локальные файлы шаблонов, находящихся в каталогах, указанных в файле settings.py, тогда как тег {% ssi %} ссылается на файлы по абсолютному пути. Однако в интересах безопасности тег {% ssi %} ограничивается множеством каталогов, перечисленных в файле settings.py, в переменной ALLOWED_INCLUDE_ROOTS.

Наконец, следует отметить, что оба тега, {% extends %} и {% include %}, принимают не только строки, но и имена переменных контекста, что позволяет динамически определять, какие шаблоны включать или наследовать.

Формы

Шаблоны отлично подходят для отображения информации, но ввод информации в базу данных – это совершенно другая задача, решение которой связано с созданием форм HTML, проверкой и сохранением введенных данных. В платформе Django имеется библиотека **форм**, которая связывает вместе три главных компонента платформы: поля базы данных, определения которых находятся в модели, теги форм HTML, отображаемые в шаблонах, и механизм проверки введенных данных и отображения сообщений об ошибках.

К моменту написания этих строк механизм форм в Django находился в переходном состоянии – библиотека, которую мы будем рассматривать здесь, известна как newforms и использует модульный подход к реализации, чем отличается в лучшую сторону от своей предшественницы, библиотеки oldforms (хотя в настоящее время инструкция import django.forms возвращает старую библиотеку). Мы будем описывать библиотеку newforms, ссылаясь на нее (и импортируя) под именем forms, в надежде, что к тому времени, когда вы будете читать эту книгу, перевод платформы Django на использование новой библиотеки обработки форм будет завершен.

Определение форм

Основу механизма обработки форм составляет класс Form, напоминающий класс Model. Подобно моделям формы по своей сути являются коллекциями объектов полей, за исключением того, что вместо таблиц в базе данных они представляют поля ввода веб-форм. В большей части случаев для нас важно, чтобы форма на 100 процентов соответствовала имеющейся модели, однако наличие отдельного класса Form обеспечивает требуемую степень гибкости в этом вопросе.

Благодаря наличию отдельного класса формы можно скрыть или опустить отдельные поля или объединить поля из нескольких классов

моделей. Конечно, иногда будет возникать необходимость обрабатывать формы, не имеющие отношение к базе данных, – и это тоже вполне осуществимо. Рассмотрим короткий пример:

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField()
    last = forms.CharField()
    middle = forms.CharField()
```

Эта форма выглядит подозрительно похожей на класс модели, определявшийся ранее, тем не менее, это совершенно самостоятельная форма, которая по воле случая имеет тот же самый набор полей (если не учитывать тот факт, что в данном случае поля являются экземплярами класса forms.Field, а не models.Field). Поля форм принимают те же аргументы, что и поля моделей.

```
class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True)
    middle = forms.CharField(max_length=100)
```

Предыдущий пример определяет форму с тремя текстовыми полями. Первые два поля, first и last, являются обязательными, поэтому, если в ходе проверки (смотрите ниже) выяснится, что они не заполнены, будет сгенерировано сообщение об ошибке. Кроме того, данное определение гарантирует, что все три поля будут иметь длину не более 100 символов. Между типами полей баз данных и типами полей форм есть много общего. Чтобы узнать, какие еще типы полей существуют, кроме тех, что вы найдете в примерах этой главы, обращайтесь к официальной документации, где вы найдете подробный перечень классов Field.

Формы, основанные на моделях

В рамках следования принципу «не повторяйся» платформа Django предоставляет возможность получить подкласс класса Form для любого класса или экземпляра модели, используя класс ModelForm. Класс ModelForm практически идентичен обычному классу Form, но в отличие от последнего имеет вложенный класс Meta (напоминающий аналогичный класс, вложенный в классы моделей), обладающий одним обязательным атрибутом model, значением которого является интересующий класс Model. Следующий пример функционально идентичен обычному классу Form, определение которого приводилось выше:

```
from django import newforms as forms
from myproject.myapp.models import Person

class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
```

Вообще говоря, для каждого класса модели обычно бывает необходимо определить хотя бы один такой класс ModelForm – даже в таких простых случаях, как этот. Этот подход демонстрирует отделение определения данных (модель) от элементов ввода данных и их проверки (форма) и обеспечивает существенную гибкость.

При использовании класса ModelForm происходит «копирование» полей класса Model в поля класса Form. Обычно такое копирование выполняется достаточно просто – поле модели CharField становится полем формы TextField или ChoiceField, если в определении поля модели имеется аргумент choices – хотя имеется несколько предупреждений, подробно описанных в таблице на официальном сайте проекта Django. Главное, что следует помнить, – поле формы считается обязательным (required=True), если соответствующее поле в модели не определено, как способное хранить пустое значение (blank=True); в противном случае поле формы становится необязательным (required=False).

Сохранение форм ModelForm

Формы, сгенерированные таким способом, имеют одно важное отличие от форм, созданных вручную, – они имеют метод save, который в случае успешного прохождения проверки сохраняет данные формы в виде записи в базе данных и возвращает объект Model. Назначение этого метода станет для вас более очевидным, когда вы прочитаете о том, как поместить информацию в форму и как выполнить ее проверку, и узнаете, что метод save позволяет пройти путь от словаря POST к созданию (или изменению) базы данных всего за несколько шагов. Продолжая предыдущий пример (подробное описание этого процесса приводится ниже в этой главе):

```
from myproject.myapp.forms import PersonForm

form = PersonForm({'first': 'John', 'middle': 'Quincy', 'last': 'Doe'})
new_person = form.save()

# Будет вызван метод __unicode__(), который выведет новый объект Person
print new_person
```

Часто возникают ситуации, когда необходимо изменить введенные данные перед тем, как они попадут в базу данных. Иногда изменения можно выполнить непосредственно в словаре POST – перед тем, как поместить его в форму, но иногда бывает проще сделать это после выполнения проверки (но до того, как данные попадут в модель). Последний способ более предпочтителен, потому что в этом случае изменение данных происходит уже после того, как данные в словаре POST превратятся в значения на языке Python.

Для обеспечения этой возможности метод save принимает необязательный аргумент commit (по умолчанию имеющий значение True), который определяет, должны ли данные действительно записываться в базу данных. При установке этого аргумента в значение False метод по-

прежнему будет возвращать объект модели, но вся ответственность за вызов метода save этого объекта целиком ляжет на ваши плечи. Следующий пример запишет данные в базу только один раз – без аргумента commit=False запись была бы произведена дважды.

```
form = PersonForm({'first': 'John', 'middle': 'Quincy', 'last': 'Doe'})

# Вернет объект Person, но запись в базу данных производиться не будет.
new_person = form.save(commit=False)

# Изменить атрибут в несохраненном объекте Person.
new_person.middle = 'Danger'

# Теперь можно сохранить информацию в базе данных.
new_person.save()
```

Другой типичный случай использования аргумента commit=False – редактирование объектов, связанных отношениями. В таких ситуациях проверка и сохранение основного объекта, а также объектов, связанных с ним, выполняются с помощью единственного объекта ModelForm. Поскольку в реляционных базах данных ссылаться можно только на строки, которые уже должны существовать, то невозможно сохранить объекты, связанные отношениями, до того, как будет сохранен основной объект.

Поэтому, когда объект ModelForm содержит информацию о связанных объектах и вы используете аргумент commit=False, платформа Django добавит в форму (*не* в объект Model!) дополнительный метод с именем save_m2m, который позволит корректно выполнить требуемую последовательность действий. В следующем примере предполагается, что модель Person устанавливает отношение типа «многие-к-многим» сама с собой.

```
# Входной объект PersonForm будет содержать "подформы"
# для дополнительных объектов Person, связанных с основным
# обьектом посредством поля ManyToManyField.
form = PersonForm(input_including_related_objects)

# В данный момент связанные обьекты не могут быть сохранены,
# поэтому фактическое их сохранение в базе данных откладывается.
new_person = form.save(commit=False)

# Изменить атрибут в несохраненном обьекте Person.
new_person.middle = 'Danger'

# После сохранения обьекта Person в БД на него можно будет
# сослаться из связанных объектов.
new_person.save()

# Поэтому теперь можно сохранить их. Не забудьте вызвать этот метод,
# в противном случае связанные объекты исчезнут!
form.save_m2m()
```

Как видите, необходимость помнить о непосредственном сохранении данных вместо того, чтобы полагаться на их отложенное сохранение, увеличивает сложность использования метода save. К счастью, эта сложность возникает не всегда, и в большинстве случаев вы можете не беспокоиться и пользоваться обычной операцией сохранения.

Отличия от модели

Иногда бывает необходимо изменить форму так, чтобы она была не совсем точной копией модели. Необходимость скрыть отдельные поля встречается достаточно часто, гораздо реже возникает потребность в массовом сокрытии или добавлении полей. Необходимость в создании обычного подкласса класса Form с чистого листа возникает очень редко, потому что существует несколько способов решить эту задачу с помощью объектов ModelForm.

Класс Meta, вложенный в класс ModelForm, позволяет определить значения пары дополнительных атрибутов, fields и exclude, которые являются обычными списками или кортежами имен включаемых или исключаемых полей (разумеется, допускается одновременно использовать только один из этих атрибутов!). Например, ниже приводится определение формы, которая позволяет вводить информацию в объект Person, исключая поле middle:

```
from django import newforms as forms
from myproject.myapp.models import Person

class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
        exclude = ('middle',)
```

Учитывая, что в модели Person определено всего три поля – first, last и middle, следующий пример использования атрибута fields в точности эквивалентен предыдущему:

```
class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
        fields = ('first', 'last')
```

Имейте в виду, что при вызове метода save такой формы он сохранит только те поля, которые ему известны. Это может вызывать проблемы, если опустить поля, которые в базе данных являются обязательными! Такие поля либо должны отмечаться, как null=True, либо для них должны определяться значения по умолчанию с помощью аргумента default.

Помимо возможности определять, какие поля модели должны отображаться, вы можете также переопределять подклассы Field на уровне формы, используемые для отображения и проверки определенных полей. Достаточно просто определить поля явно, как это делалось в начале

данного раздела главы, и эти определения будут преобладать перед определениями, выполненными в модели. Это может потребоваться, когда необходимо изменить аргументы, передаваемые объектам Field на уровне формы, такие как max_length или required, или изменить сам класс поля (например, вынудить TextField отображаться как CharField или добавить аргумент choices в определение поля CharField, чтобы превратить его в ChoiceField). Например, следующий фрагмент просто уменьшает поле ввода имени, делая его короче обычного:

```
class PersonForm(forms.ModelForm):
    first = forms.CharField(max_length=10)

    class Meta:
        model = Person
```

Специального упоминания заслуживают поля, устанавливающие отношения в форме, – ModelChoiceField и ModelMultipleChoiceField, которые соответствуют полям ForeignKey и ManyToManyField. Несмотря на то, что имеется возможность использовать аргумент limit_choices_to при определении полей на уровне модели, вы также можете использовать аргумент queryset при определении полей на уровне формы, который, как и следовало ожидать, принимает определенный объект QuerySet. В этом случае вы получаете возможность переопределить любые подобные ограничения (или определить новые ограничения), установленные на уровне модели, и определить ограничения на уровне класса ModelForm, как показано в следующем примере, где предполагается, что модель Person не накладывает ограничений на поле parent типа ForeignKey:

```
# Обычная форма без ограничений (так как модель не накладывает
# ограничений на поле 'parent')
class PersonForm(forms.ModelForm):
    class Meta:
        model = Person

# Форма выбора людей из семьи Smith (чьи родители носили фамилию Smith)
class SmithChildForm(forms.ModelForm):
    parent = forms.ModelChoiceField(queryset=Person.objects.filter(
                                                    last='Smith'))
    class Meta:
        model = Person
```

Создание подклассов форм

Во многих ситуациях, когда требуется использовать классы Form или ModelForm, можно воспользоваться преимуществами объектно-ориентированной природы языка Python, чтобы избежать ненужных повторений. Подклассы класса Form сами тоже могут иметь подклассы, которые будут обладать всеми полями, определяемыми в родительском классе. Например:

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True)
    middle = forms.CharField(max_length=100)

class AgedPersonForm(PersonForm):
    # наследует поля first, last, middle
    age = forms.IntegerField()

class EmployeeForm(PersonForm):
    # наследует поля first, last, middle
    department = forms.CharField()

class SystemUserForm(EmployeeForm):
    # наследует поля first, last, middle и department
    username = forms.CharField(maxlength=8, required=True)
```

Кроме того, можно выполнить так называемое смешивание классов, используя механизм множественного наследования.

```
class BookForm(forms.Form):
    title = forms.CharField(max_length=100, required=True)
    author = forms.CharField(max_length=100, required=True)

class InventoryForm(forms.Form):
    location = forms.CharField()
    quantity = forms.IntegerField()

class BookstoreBookForm(BookForm, InventoryForm):
    # обладает полями title, author, location и quantity
    pass
```

Обратите внимание: применяя такой подход к подклассам класса ModelForm, возможно также изменять атрибуты класса Meta, изменяя или добавляя значения fields или exclude, чтобы ограничить круг доступных полей.

Заполнение форм

Экземпляр любой формы из библиотеки форм платформы Django может быть **связанным**, то есть иметь некоторые данные, ассоциированные с формой, или **несвязанным**, то есть пустым. Несвязанные, пустые формы в первую очередь предназначены для создания пустых форм HTML, заполняемых пользователем, а так как отсутствует возможность выполнить их проверку (если форма пустая, то нежелательный ввод весьма маловероятен), у вас едва ли появится желание сохранять их содержимое в базе данных. Связанные формы – это место, где выполняется основной объем действий.

Привязка данных к форме выполняется в момент ее создания, и после создания изменить форму будет невозможно. Такое решение может показаться негибким, но это делает процесс использования формы более

явным и независимым. Кроме того, при таком подходе ликвидируется любая неоднозначность состояния механизма проверки формы, данные в которой были изменены (как это было возможно в библиотеке oldforms).

Давайте создадим связанную форму на основе класса PersonForm, наследующего класс ModelForm, добавив сначала то, что впоследствии может превратиться в функцию представления, выполняющую обработку формы. Вызывать метод request.POST.copy() необязательно, но в этом есть определенный смысл, потому что в этом случае вы получаете возможность изменять свою копию словаря, оставляя оригинальное содержимое запроса нетронутым, – на случай, если оно понадобится позже.

```
from myproject.myapp.forms import PersonForm

def process_form(request):
    post = request.POST.copy() # например, {'last': 'Doe', 'first': 'John'}
    form = PersonForm(post)
```

Следует заметить, что наличие в словаре с данными формы лишних пар ключ/значение не будет вызывать проблем – формы просто игнорируют любые входные данные, не соответствующие определенному набору полей. Это означает, что можно взять словарь POST, полученный от более крупной формы, и использовать его для заполнения объекта Form, содержащего подмножество полей, имеющихся в словаре.

Имеется также возможность создавать формы, которые хотя и несвязаны, но загружаются с начальными данными, отображаемыми, когда форма печатается в виде шаблона. Удачно названный аргумент initial конструктора – это словарь, который в виде позиционного аргумента используется для привязки данных. Отдельные поля форм имеют аналогичный параметр, позволяющий указывать их собственные значения по умолчанию, которые в случае возникновения противоречия преодолеваются значениями словаря уровня формы.

Ниже приводится пример создания измененной версии формы PersonForm, в которой поле фамилии предварительно заполняется значением «Smith» (внутри определения формы), а поле имени – значением «John» (во время выполнения, когда создается экземпляр формы). Пользователи, конечно же, могут переопределить эти значения при заполнении формы.

```
from django import newforms as forms
from django.shortcuts import render_to_response

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True, initial='Smith')
    middle = forms.CharField(max_length=100)

def process_form(request):
    if not request.POST: # Отобразить форму, если ничего не было отправлено.
```

```
    form = PersonForm(initial={'first': 'John'})
    return render_to_response('myapp/form.html', {'form': form})
```

Примечание

Если бы в этом примере при создании экземпляра PersonForm аргумент initial имел значение – например {'first': 'John', 'last': 'Doe'}, то значение «Doe» (для ключа 'last'), указанное на уровне экземпляра, переопределило бы значение «Smith», указанное на уровне класса.

Главное преимущество аргумента initial заключается в том, что его значение может конструироваться во время создания формы. Например, использование этого аргумента позволит сослаться на информацию, недоступную на момент определения формы или модели, – обычно эта информация находится в объекте запроса.

Посмотрим, как это обстоятельство можно использовать в функции представления, которая имеет отношение к добавлению новых записей Person, связанных с другими подобными записями. Представим, что в нашей модели Person появилось новое поле parent типа ForeignKey, ссылающееся на саму модель Person, и для модели Person определена простая форма ModelForm.

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

# Адрес URL представления: /person/<id>/children/add/
def add_relative(request, **kwargs):
    # Отобразить форму, если ничего не было отправлено
    if not request.POST:
        relative = get_object_or_404(Person, pk=kwargs['id'])
        form = PersonForm(initial={'last': relative.last})
        return render_to_response('person/form.html', {'form': form})
```

Для краткости мы опустили обработку отправленной формы, которая наверняка присутствовала бы в действующей функции представления. Обратите внимание, что мы получили объект relative на основе значений в URL и передали содержимое поля last этого объекта в качестве начального значения поля last формы. Другими словами, мы организовали форму так, что при вводе информации о детях поле фамилии автоматически заполняется фамилией родителей, что может быть удобно, если вашим пользователям придется вводить большое число записей.

Проверка и очистка

Формы, как правило, не обладают информацией о состоянии, поэтому они требуют наличия некоторого механизма запуска выполнения проверки данных, с которыми они связаны, – если они вообще связаны (проверка и очистка не применяются к несвязанным формам). Чтобы вынудить форму запустить процедуру проверки, можно явно вызвать

явным и независимым. Кроме того, при таком подходе ликвидируется любая неоднозначность состояния механизма проверки формы, данные в которой были изменены (как это было возможно в библиотеке oldforms).

Давайте создадим связанную форму на основе класса PersonForm, наследующего класс ModelForm, добавив сначала то, что впоследствии может превратиться в функцию представления, выполняющую обработку формы. Вызывать метод request.POST.copy() необязательно, но в этом есть определенный смысл, потому что в этом случае вы получаете возможность изменять свою копию словаря, оставляя оригинальное содержимое запроса нетронутым, – на случай, если оно понадобится позже.

```
from myproject.myapp.forms import PersonForm

def process_form(request):
    post = request.POST.copy() # например, {'last': 'Doe', 'first': 'John'}
    form = PersonForm(post)
```

Следует заметить, что наличие в словаре с данными формы лишних пар ключ/значение не будет вызывать проблем – формы просто игнорируют любые входные данные, не соответствующие определенному набору полей. Это означает, что можно взять словарь POST, полученный от более крупной формы, и использовать его для заполнения объекта Form, содержащего подмножество полей, имеющихся в словаре.

Имеется также возможность создавать формы, которые хотя и несвязаны, но загружаются с начальными данными, отображаемыми, когда форма печатается в виде шаблона. Удачно названный аргумент initial конструктора – это словарь, который в виде позиционного аргумента используется для привязки данных. Отдельные поля форм имеют аналогичный параметр, позволяющий указывать их собственные значения по умолчанию, которые в случае возникновения противоречия преодолеваются значениями словаря уровня формы.

Ниже приводится пример создания измененной версии формы PersonForm, в которой поле фамилии предварительно заполняется значением «Smith» (внутри определения формы), а поле имени – значением «John» (во время выполнения, когда создается экземпляр формы). Пользователи, конечно же, могут переопределить эти значения при заполнении формы.

```
from django import newforms as forms
from django.shortcuts import render_to_response

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True, initial='Smith')
    middle = forms.CharField(max_length=100)

def process_form(request):
    if not request.POST: # Отобразить форму, если ничего не было отправлено.
```

```
    form = PersonForm(initial={'first': 'John'})
    return render_to_response('myapp/form.html', {'form': form})
```

Примечание

Если бы в этом примере при создании экземпляра PersonForm аргумент initial имел значение – например {'first': 'John', 'last': 'Doe'}, то значение «Doe» (для ключа 'last'), указанное на уровне экземпляра, переопределило бы значение «Smith», указанное на уровне класса.

Главное преимущество аргумента initial заключается в том, что его значение может конструироваться во время создания формы. Например, использование этого аргумента позволит сослаться на информацию, недоступную на момент определения формы или модели, – обычно эта информация находится в объекте запроса.

Посмотрим, как это обстоятельство можно использовать в функции представления, которая имеет отношение к добавлению новых записей Person, связанных с другими подобными записями. Представим, что в нашей модели Person появилось новое поле parent типа ForeignKey, ссылающееся на саму модель Person, и для модели Person определена простая форма ModelForm.

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

# Адрес URL представления: /person/<id>/children/add/
def add_relative(request, **kwargs):
    # Отобразить форму, если ничего не было отправлено
    if not request.POST:
        relative = get_object_or_404(Person, pk=kwargs['id'])
        form = PersonForm(initial={'last': relative.last})
        return render_to_response('person/form.html', {'form': form})
```

Для краткости мы опустили обработку отправленной формы, которая наверняка присутствовала бы в действующей функции представления. Обратите внимание, что мы получили объект relative на основе значений в URL и передали содержимое поля last этого объекта в качестве начального значения поля last формы. Другими словами, мы организовали форму так, что при вводе информации о детях поле фамилии автоматически заполняется фамилией родителей, что может быть удобно, если вашим пользователям придется вводить большое число записей.

Проверка и очистка

Формы, как правило, не обладают информацией о состоянии, поэтому они требуют наличия некоторого механизма запуска выполнения проверки данных, с которыми они связаны, – если они вообще связаны (проверка и очистка не применяются к несвязанным формам). Чтобы вынудить форму запустить процедуру проверки, можно явно вызвать

метод is_valid, возвращающий логическое значение, или один из методов отображения (как показано ниже), каждый из которых также неявно выполняет проверку.

Давайте переделаем нашу предыдущую функцию представления add_relative, выполняющую обработку формы, так, чтобы наряду с отображением пустой формы она обрабатывала введенные данные. Для этого придется немного изменить логику и добавить гибкости в обычную идиому Django, которая проверяет существование словаря POST и производит проверку данных (или генерирует пустую форму) и в случае ошибки отображает форму. Таким образом, форма будет отображаться либо в ответ на запрос, не содержавший данные формы, либо в случае выявления ошибки в процессе проверки.

```
# Адрес URL представления: /person/<id>/children/add/
def add_relative(request, **kwargs):
    # Получить данные о родителе

    # Выполнить проверку, если пользователем была отправлена форма с данными
    if request.POST:
        form = PersonForm(request.POST)
        if form.is_valid:
            new_person = form.save()
            return HttpResponseRedirect(new_person.get_absolute_url())
        # В противном случае подготовить пустую форму,
        # предварительно заполнив поле фамилии
        else:
            relative = get_object_or_404(Person, pk=kwargs['id'])
            form = PersonForm(initial={'last': relative.last})
        # Отобразить форму в ответ на запрос, не содержавший
        # данные формы, или в случае ошибки.
        # Сообщения об ошибках, если таковые имеются,
        # будут отображены шаблоном.
        return render_to_response('person/form.html', {'form': form})
```

После выполнения проверки объект формы обретает один из двух новых атрибутов: errors — словарь с сообщениями об ошибках или cleaned_data — словарь, содержащий «очищенные» версии значений, изначально привязанных к форме. Объект формы никогда не будет иметь оба атрибута сразу, так как атрибут cleaned_data создается, только когда проверка завершается успехом, а атрибут errors — естественно, когда проверка терпит неудачу.

Словарь errors имеет очень простой формат — ключами являются имена полей, а значениями — списки строк (каждая строка представляет собой текст сообщения, поясняющего причины, почему проверка потерпела неудачу). Словарь errors обычно содержит пары ключ/значения только для тех полей, где были обнаружены ошибки. Ниже в этой главе мы рассмотрим некоторые вспомогательные методы, которые предоставляются объектом Form, чтобы упростить отображение этих сообщений об ошибках.

В основе концепции «чистых» данных лежит необходимость **нормализации** входной информации – преобразования их из нескольких возможных форматов ввода в единый выходной формат, соответствующий формату проверки и хранения в базе данных. Например, формы со связанными данными, поступающими непосредственно из словаря request.POST, обычно содержат строки, поэтому процедура «очистки» числовых полей должна выполнить приведение этих строк к типу int или long; строковые значения для полей с датами, такие как «2007-10-29», должны быть преобразованы в объекты datetime и т. д.

Нормализация в основном требуется для корректной работы методов автоматической проверки и сохранения, но, кроме того, благодаря ей любой программный код на языке Python, взаимодействующий с содержимым формы, будет получать доступ к корректным типам данных. Если вам потребуется получить доступ к оригинальным, ненормализованным значениям, можно обратиться к атрибуту data формы.

Отображение форм

Объекты форм обладают несколькими полезными методами, позволяющими отображать их в различных предопределенных форматах HTML без тегов <form> и кнопки отправки, но включая теги <label>. Кроме того, если вам необходим более полный контроль над отображением формы, имеется возможность отобразить каждое поле формы в отдельности. Наконец, следует заметить, что хотя эти методы обычно используются в шаблонах, тем не менее, это совершенно необязательно и в случае необходимости вы можете вызывать любые из этих методов из программного кода на языке Python.

Каждое поле формы в платформе Django знает, как отображать себя в виде одного или нескольких тегов HTML, и это их поведение может быть изменено с помощью **виджетов**, рассматриваемых в конце этой главы. Кроме того, значения атрибутов name и id этих тегов, а также значения атрибутов for соответствующих тегов <label> создаются из определяемых вами имен полей атрибутов класса Form. Текст внутри тегов <label> по умолчанию генерируется из имен полей, в которых символы подчеркивания замещаются пробелами, первые символы слов приводятся к верхнему регистру, а в конец добавляется символ :, если имя поля не оканчивается знаком пунктуации.

Ниже приводится пример, который поможет вам понять назначение некоторых параметров и их влияние на выходную разметку HTML. Для начала снова приведем созданное вручную определение формы PersonForm:

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True, initial='Smith')
```

```
middle = forms.CharField(max_length=100)
```

и покажем, как выглядит разметка HTML для поля first, когда форма отображается как часть таблицы:

```
<tr><th><label for="id_first">First:</label></th><td><input id="id_first"
type="text" name="first" maxlength="100" /></td></tr>
```

Имеется возможность изменить поведение атрибутов id и тегов <label>, добавив в вызов конструктора формы аргумент auto_id: значение False в этом аргументе вообще предотвращает отображение атрибутов id и тегов <label>; значение True обеспечивает их создание, как было показано выше; а если аргумент содержит строку формата, такую как 'id_s%', то строка, подставляемая на место шаблонного символа в строке форматирования, используется вместо имени атрибута. Кроме того, завершающий символ : в метках можно переопределить с помощью аргумента label_suffix, который является обычной строкой.

Ниже показано, как можно создать экземпляр PersonForm с выключенным аргументом auto_id и с «выключенным» аргументом label_suffix, за счет передачи в нем пустой строки:

```
pf = PersonForm(auto_id=False, label_suffix='')
```

В этом случае при отображении формы разметка HTML для поля first будет выглядеть так:

```
<tr><th>First</th><td><input type="text" name="first" maxlength="100" />
td></tr>
```

Наконец, ту же форму можно вывести, использовав собственные строковые значения в аргументах auto_id и label_suffix:

```
pf = PersonForm(auto_id='%s_id', label_suffix='?')
```

Это в результате даст следующую разметку:

```
<tr><th><label for="first_id">First?</label></th><td><input id="first_id"
type="text" name="first" maxlength="100" /></td></tr>
```

Как видите, автоматический механизм платформы Django вывода формы обеспечивает высокую гибкость. В следующем разделе мы покажем, как фактически получить такой вывод из объекта формы.

Отображение форм целиком

По умолчанию при выводе формы используется ее собственный метод as_table, который выводит форму в виде строк таблицы, состоящих из двух полей, описанных с помощью тегов <tr> и <td>, но при этом, чтобы обеспечить большую гибкость, опускаются теги <table>. Помимо метода as_table имеется также метод as_p — он выводит форму с использованием тегов параграфов, и метод as_ul, который использует теги элементов списков (но обычно опускает обертывающие теги самого списка <ul>).

Примечание

При выводе форм опускаются «внешние» теги, такие как <table></table>, потому что их включение может осложнить внедрение всей формы в разметку HTML шаблона. То же самое относится к кнопке, выполняющей отправку формы – во многих шаблонах предусматриваются иные методы отправки, такие как <input type="button" /> или <input type="submit" />, поэтому Django оставляет возможность окончательного решения за вами.

Кроме того, при таком способе отображения формы автоматически выводятся все существующие сообщения об ошибках: рядом с соответствующим полем, в зависимости от используемого метода вывода, отображается тег <ul> с одним или более тегами <li>. Методы as_table и as_ul выводят список ошибок в том же теге, где находится само поле (внутри тегов <td> и <li>, соответственно), а метод as_p создает для вывода списков ошибок новые параграфы. В любом случае сообщения об ошибках выводятся, как показано ранее, перед элементами форм.

Имеется возможность изменить порядок вывода списков с сообщениями об ошибках, создав свой подкласс от класса django.forms.util.ErrorList и передав его в виде аргумента error_class методу вывода формы. А если вам необходимо изменить порядок следование полей/списков сообщений, просто измените их порядок следования в классе Form – вот и все.

Отображение форм по частям

В дополнение к вспомогательным методам, упомянутым выше, имеется возможность более точного управления отображением форм. Отдельные объекты Field доступны в виде ключей словаря самой формы, что позволяет отображать их как нужно и где нужно. Вы можете также выполнять итерации по самой форме, используя особенность «утиной» типизации в языке Python. Независимо от того, каким способом будет получен доступ к полям, у каждого из них имеется собственный атрибут errors – список строк, представляющий тот же неупорядоченный список, который был ранее представлен в методах вывода всей формы (и переопределяемый точно так же).

В библиотеке oldforms самый простой способ переопределить представление по умолчанию полей формы в разметке HTML заключался в том, чтобы получить доступ к атрибуту data поля и обернуть его собственной разметкой HTML – этот прием также возможен и при использовании библиотеки newforms. Однако мощь виджетов, на наш взгляд, уменьшает потребность в этом приеме.

Виджеты

Виджет на языке форм платформы Django – это объект, который знает, как отобразить элемент формы в разметку HTML. В дополнение

к подклассам Field полей моделей и подклассам Field полей форм платформа Django предоставляет приличную библиотеку подклассов Widget виджетов. Каждому полю формы в соответствие поставлен определенный подкласс Widget виджета, посредством которого отображаются данные в поле, когда приходит время отображать форму в шаблон. Например, для отображения полей типа CharField по умолчанию используется подкласс класса Widget с именем TextInput, который просто отображает разметку <input type="text" />.

Часто бывает вполне достаточно зависимости поле-виджет по умолчанию, когда сами виджеты даже незаметны. Однако бывают случаи, когда возникает необходимость изменять атрибуты виджетов или замещать одни виджеты другими. Первый случай является более типичным, так как позволяет изменять атрибуты разметки HTML для поля. В следующем примере изменяется значение атрибута «size» обычного текстового поля:

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True)
    middle = forms.CharField(max_length=100,
        widget=forms.TextInput(attrs={'size': 3}
    )
```

При использовании этой формы поле middle будет отображаться, как показано ниже:

```
<input id="id_middle" maxlength="100" type="text" name="middle" size="3" />
```

Как видите, модификации такого вида возможны благодаря тому, что подклассы класса Widget (такие как TextInput) принимают словарь attrs, отображаемый непосредственно в атрибуты тегов HTML. В данном случае мы не ограничиваем фактический размер вводимой строки (пользователь по-прежнему может ввести до 100 символов), но мы ограничиваем видимый размер поля.

Переопределение виджета, используемого по умолчанию

Параметр widget конструктора подкласса Field может также использоваться для замены виджета, используемого по умолчанию, – достаточно лишь указать другой подкласс виджета. Например, вместо виджета TextInput можно было использовать виджет TextArea. Эта особенность полей форм обеспечивает отделение визуального представления поля (виджета) от алгоритма его проверки (поля формы). Кроме того, это означает, что вы можете определять собственные подклассы класса Widget, если встроенные виджеты не отвечают вашим потребностям.

Несмотря на то, что описание создания виджетов с самого начала выходит далеко за рамки темы этой главы, тем не менее, мы поделимся простым способом, позволяющим быстро создавать подклассы класса

Widget. Если вы вдруг обнаружите, что часто используете словарь attrs для какого-то определенного типа виджетов, вы можете создать подкласс от интересующего вас класса виджетов и присвоить ему словарь attrs со значениями по умолчанию.

```
from django import newforms as forms

class LargeTextareaWidget(forms.Textarea):
    def __init__(self, *args, **kwargs):
        kwargs.setdefault('attrs', {}).update({'rows': 40, 'cols': 100})
        super(LargeTextareaWidget, self).__init__(*args, **kwargs)
```

В этом примере используется один интересный прием – метод setdefault действует так же, как метод get, то есть возвращает существующее значение заданного ключа или указанное значение, если ключ еще не существует. Однако кроме этого он еще и изменяет словарь, сохраняя в нем указанное значение. Этот метод используется здесь, чтобы гарантировать наличие словаря attrs в словаре kwargs именованных аргументов независимо от фактических аргументов конструктора. После этого словарь attrs дополняется необходимыми значениями по умолчанию с помощью метода update.

Получающийся в результате новый виджет LargeTextArea ведет себя как обычный виджет TextArea, но по умолчанию всегда имеет размеры в 40 строк и 100 символов в строке. Теперь мы сможем использовать наш новый виджет для отображения всех полей, которые требуется отобразить в виде областей ввода большого размера. Для следующего примера предположим, что в локальном файле forms.py приложения мы сохранили собственные классы форм.

```
from django import newforms as forms
from myproject.myapp.forms import LargeTextareaWidget

class ContentForm(forms.Form):
    name = forms.CharField()
    markup = forms.ChoiceField(choices=[
        ('markdown', 'Markdown'),
        ('textile', 'Textile')
    ])
    text = forms.Textarea(widget=LargeTextareaWidget)
```

Конечно, у нас имеется возможность пойти еще дальше. Так как аргумент widget в подклассе поля просто устанавливает значение атрибута widget, мы можем создать новый подкласс поля, в котором всегда будет использоваться наш собственный виджет.

```
class LargeTextareaWidget(forms.Textarea):
    def __init__(self, *args, **kwargs):
        kwargs.setdefault('attrs', {}).update({'rows': 40, 'cols': 100})
        super(LargeTextareaWidget, self).__init__(*args, **kwargs)

class LargeTextarea(forms.Field):
    widget = LargeTextareaWidget
```

Теперь мы можем изменить предыдущий пример создания формы и задействовать наш собственный подкласс поля.

```
class ContentForm(forms.Form):
    name = forms.CharField()
    markup = forms.ChoiceField(choices=[
        ('markdown', 'Markdown'),
        ('textile', 'Textile')
    ])
    text = LargeTextarea()
```

Как и в других ситуациях, тот факт, что платформа Django – это исключительно язык программирования Python, означает, что вы легко можете менять различные классы и объекты, как показано в данном примере, – когда в этом возникнет необходимость. Это знание поможет вам в любых других областях, где потребуется выполнить настройку поведения отдельных аспектов платформы.

В заключение

В этой главе вы узнали о синтаксисе языка шаблонов Django, о том, как эти шаблоны отображаются с учетом содержимого контекстных словарей, а также познакомились с более сложными темами, такими как наследование и включение шаблонов. Кроме того, вы узнали, как воспроизводить формы – независимые или представляющие определенные классы моделей, как заставить их выполнить проверку данных и отобразить себя в виде разметки HTML. В заключение мы рассмотрели некоторые возможности настройки, предоставляемые виджетами.

Эта глава завершает вторую часть книги «Подробно о Django», и теперь у вас имеется приличная база знаний о том, что предлагает платформа Django, – начиная от определения моделей, адресов URL и обработки запросов и заканчивая сведениями о шаблонах и формах. В следующих четырех главах третьей части книги «Приложения Django в примерах», приводятся примеры приложений, где используются сведения, которые мы уже рассмотрели, вводятся некоторые новые понятия и дополняются уже изученные.

III

Приложения Django в примерах

7. Фотогалерея
8. Система управления содержимым
9. Живой блог
10. Pastebin

7

Фотогалерея

Типичной функциональностью многих веб-сайтов, управляемых данными, является предоставляемая пользователям возможность добавлять не только текст, но и файлы – офисные документы, видеоролики, документы PDF и, конечно же, вездесущие изображения. В приводимом примере приложения мы продемонстрируем работающий пример применения поля выгрузки файлов изображений на сервер – `ImageField` – в несложном приложении-галерее. Кроме того, мы создадим наш собственный подкласс класса `ImageField`, позволяющий автоматически генерировать миниатюры изображений. Наконец, мы предусмотрели возможность динамического изменения корневого адреса URL приложения, чтобы максимально упростить его развертывание.

Приложение, которое мы создадим, является упрощенной коллекцией универсальных объектов `Item`, каждый из которых может быть связан с несколькими объектами класса `Photo`. Вся эта небольшая иерархия будет представлена в виде проекта `gallery` на платформе Django с приложением `items` внутри него (за неимением более привлекательного названия).

Это приложение можно расширить и создать на его основе полноценный сайт фотогалереи, где наш объект `Item` станет, скорее всего, контейнером или папкой, используемой исключительно для организации фотографий. На базе этого приложения можно также создать приложение некоего демонстрационного зала, где каждый объект `Item` обладает дополнительными атрибутами (такими как модель автомобиля, производитель и год выпуска), обеспечивающими более разнообразные возможности. В нашем же примере каждый объект `Item` можно считать отдельным фотоальбомом.

Мы не будем делать ничего лишнего, поэтому в нашем приложении везде, где только возможно, будут использоваться универсальные

представления, а весь ввод данных будет производиться исключительно с помощью приложения администрирования, встроенного в платформу Django. Структура приложения выглядит очень просто:

- Статическое приветствие в шаблоне главной (индексной) страницы.
- На главной странице отображается «стенд» (небольшое подмножество миниатюр фотографий).
- На странице со списком отображаются все фотоальбомы (объекты `Item`), имеющиеся на сайте.
- Представление, предназначенное для просмотра содержимого одного фотоальбома (объекта `Item`), выводит список всех фотографий – объектов `Photo` (опять же в виде миниатюр).
- Представление, предназначенное для просмотра одной фотографии (объект `Photo`), отображает изображение в полном разрешении.

Мы начнем с определения модели, а затем выполним действия, которые необходимы, чтобы обеспечить возможность выгрузки фотографий на сервер с помощью приложения администрирования. Вслед за этим подробно рассмотрим создание нашего собственного поля модели. В заключение применим принцип «не повторяйся» к набору адресов URL и создадим шаблоны для отображения миниатюр и изображений в остальной части сайта.

Примечание

В этом примере предполагается наличие установленного веб-сервера Apache + `mod_python`, хотя можно использовать и иные стратегии развертывания. Сервер разработки Django на практике не годится для работы этого приложения, так как галерея предполагает работу с большим числом статических файлов, таких как изображения. Дополнительно о настройке Apache можно прочитать в приложении В «Установка и запуск Django».

Модель

Ниже приводится содержимое файла `models.py` приложения. За исключением одного изменения, которое будет внесено позже, – это окончательная версия. Обратите внимание на применение декоратора `@permalink` к методам `get_absolute_url` – этот декоратор будет рассматриваться в конце главы.

```
class Item(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()

    class Meta:
        ordering = ['name']

    def __unicode__(self):
        return self.name
```

```
    @permalink
    def get_absolute_url(self):
        return ('item_detail', None, {'object_id': self.id})

class Photo(models.Model):
    item = models.ForeignKey(Item)
    title = models.CharField(max_length=100)
    image = models.ImageField(upload_to='photos')
    caption = models.CharField(max_length=250, blank=True)

    class Meta:
        ordering = ['title']

    def __unicode__(self):
        return self.title

    @permalink
    def get_absolute_url(self):
        return ('photo_detail', None, {'object_id': self.id})

class PhotoInline(admin.StackedInline):
    model = Photo

class ItemAdmin(admin.ModelAdmin):
    inlines = [PhotoInline]

admin.site.register(Item, ItemAdmin)
admin.site.register(Photo)
```

Как видите, объект Item достаточно прост, а гвоздем программы является объект Photo – он имеет не только поле связи с родительским объектом Item, но еще поля заголовка, изображения и необязательное поле подписи. Оба объекта регистрируются в приложении администрирования, кроме того, у обоих имеется атрибут Meta.ordering.

Основное наше внимание будет уделено полю ImageField в модели Photo, потому что это поле мы будем по ходу дела изменять и оно требует некоторой настройки – в отличие от большей части других полей модели. А теперь посмотрим, как подготовить приложение к работе.

Подготовка к выгрузке файлов

Прежде чем мы сможем выгружать файлы на наш сайт галереи, нам необходимо определить место, где эти файлы будут сохраняться. Поля FileField и ImageField сохраняют выгружаемые данные в подкаталоге, имя которого определяется аргументом upload_to поля и который расположен в каталоге, определяемом параметром настройки MEDIA_ROOT в файле settings.py. В реализации нашей модели мы используем аргумент upload_to со значением 'photos', поэтому, если в файле settings.py определено следующее:

```
MEDIA_ROOT = '/var/www/gallery/media/'
```

наши фотографии будут попадать в каталог /var/www/gallery/media/photos/. Если этот каталог не существует, его нужно создать, а также сделать его доступным для записи пользователю или группе, с правами которых выполняется веб-сервер. В нашей системе Debian веб-сервер Apache выполняется с правами пользователя www-user, поэтому короткий сеанс работы с командной оболочкой у нас выглядел, как показано ниже (за дополнительной информацией об использовании командной строки обращайтесь к приложению А «Основы командной строки»):

```
user@example:~ $ cd /var/www/gallery/media
user@example:/var/www/gallery/media $ ls
admin
user@example:/var/www/gallery/media $ mkdir photos
user@example:/var/www/gallery/media $ ls -l
total 4
lrwxrwxrwx 1 root  root    59 2008-03-26 21:41 admin ->
/usr/lib/python2.4/site-packages/django/contrib/admin/media
drwxrwxr-x 2 user user 4096 2008-03-26 21:44 photos
user@example:/var/www/gallery/media $ chgrp www-data photos
user@example:/var/www/gallery/media $ chmod g+w photos
user@example:/var/www/gallery/media $ ls -l
total 4
lrwxrwxrwx 1 root  root    59 2008-03-26 21:41 admin ->
/usr/lib/python2.4/site-packages/django/contrib/admin/media
drwxrwxr-x 2 user www-data 4096 2008-03-26 21:44 photos
```

Предыдущие действия можно будет выполнить, только если учетная запись, с правами которой вы работаете, принадлежит группе www-data, – в зависимости от настроек системы вам может потребоваться воспользоваться командой sudo или использовать какой-нибудь похожий прием. Мы посчитали, что поскольку нам часто приходится решать задачи, связанные с администрированием веб-сервера, будет очень удобно добавить свою учетную запись в его группу – пока каталоги и файлы будут доступны группе для записи (как показано выше), с ними смогут взаимодействовать и веб-сервер, и мы, под нашей учетной записью.

Наконец, следует учесть, что более полное приложение наверняка имело бы в каталоге media еще пару символических ссылок, – в конце концов, вам потребовалось бы где-то хранить свои таблицы стилей CSS и сценарии JavaScript, а также настроить веб-сервер, чтобы он мог обслуживать эти каталоги. Например, при использовании библиотеки mod_python с сервером Apache вам необходимо определить короткий блок с параметрами настройки, создающий «разрыв» в адресном пространстве URL, обслуживаемом платформой Django, чтобы эти статические файлы могли обслуживаться непосредственно веб-сервером Apache. Подробнее о настройке mod_python рассказывается в приложении В.

Установка PIL

К настоящему моменту мы уже добавили свое приложение в файл settings.py, выполнили команду syncdb и почти готовы выгружать изображения на сервер. Однако, как вы увидите через минуту, осталось выполнить еще одно заключительное действие. Если теперь попробовать открыть в броузере административный раздел сайта в нынешнем состоянии приложения (мы настроили проект Django и каталог, куда будут выгружаться фотографии), то, скорее всего, вы увидите страницу, сходную с изображенной на рис. 7.1.

Другими словами, для работы с полями ImageField необходима специальная библиотека для языка Python – PIL, или Python Imaging Library, обычно применяемая для выполнения разнообразных действий с изображениями. Поле ImageField использует ее, чтобы убедиться, что выгружаемый файл действительно является изображением, и сохранить его высоту и ширину, если используются необязательные параметры height_field и width_field, – мы будем применять их для создания миниатюр.

Чтобы установить библиотеку PIL в UNIX-подобных системах, типа Linux или Mac, загрузите комплект исходных текстов (Source Kit) по адресу *http://www.pythonware.com/products/pil/* и запустите команду setup.py install в каталоге установки после распаковывания комплекта.

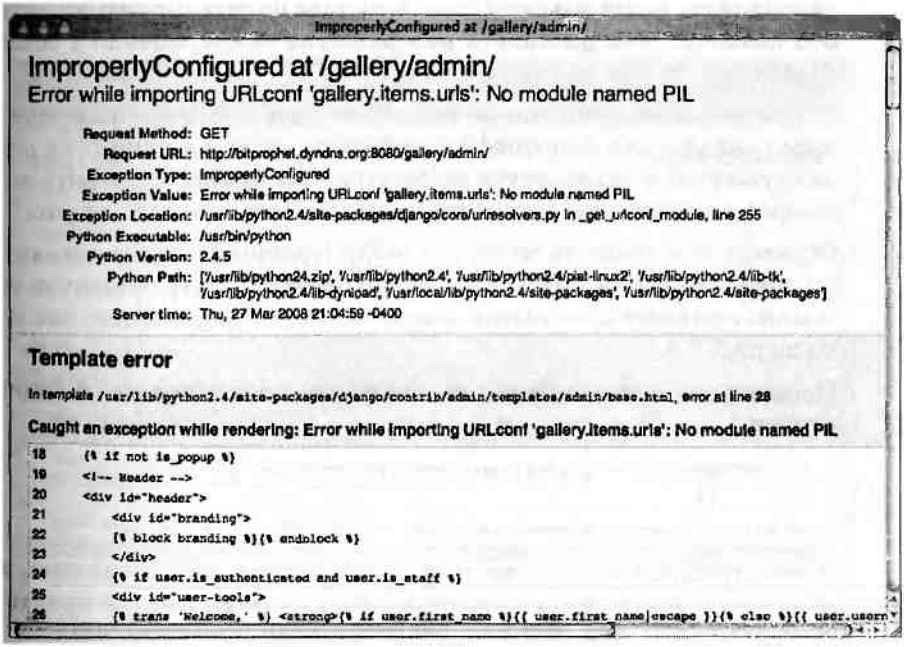

Рис. 7.1. Что происходит, когда в системе не установлена библиотека PIL

Для Win32 необходимо загрузить установочный файл .exe, соответствующий вашей версии Python, и установить его.

Самый простой способ установить пакеты Python сторонних разработчиков на любой платформе – использовать утилиту Easy Install. Она сама побеспокоится обо всех зависимостях и выполнит загрузку и установку за один шаг. Чтобы осуществить все описанные выше действия, достаточно просто выполнить команду easy_install pil. Подробную информацию о том, как получить и использовать утилиту Easy Install, вы найдете на сайте PEAK Developer's Center, по адресу *http://peak.telecommunity.com/DevCenter/EasyInstall*.

Проверка поля ImageField

Независимо от того, как была выполнена установка библиотеки PIL, сразу же после этого следует перезапустить веб-сервер; все ошибки, связанные с отсутствием библиотеки PIL, должны исчезнуть. После этого можно двинуться дальше и проверить работу нашего поля ImageField.

Из-за особенностей строения модели нашей галереи мы не можем просто взять и выгрузить на сервер произвольный файл изображения, если он не связан с объектом Item. При внимательном рассмотрении определения модели можно увидеть, что мы определили административную и встроенную версии классов, благодаря чему можем редактировать наши объекты Photo в составе родительского объекта Item. Это позволит нам добавлять изображения одновременно с созданием объектов Item, как показано на рис. 7.2.

После сохранения нового объекта Item выбранное изображение (в нашем случае – это фотография любимого кролика одного из авторов) выгружается и сохраняется на сервере. Мы можем проверить это с помощью приложения администрирования, как показано на рис. 7.3.

Обратите внимание на ссылку Currently: (текущий), расположенную выше кнопки выбора файла для первого объекта Photo, – щелчок на этой ссылке приведет к открытию выгруженного изображения, как показано на рис. 7.4.

Проверить наличие выгруженного файла можно также из командной строки.

```
user@example:/var/www/gallery/media/photos $ ls -l
total 144
-rw-r-r- 1 www-data www-data 140910 2008-03-27 21:26 IMG_0010.jpg
```

Нам потребовалось дать некоторые пояснения, но, как видите, процедура настройки функции выгрузки изображения прошла практически безболезненно. Все, что нам действительно пришлось сделать, – это определить модель, создать каталог для хранения изображений и установить библиотеку для работы с изображениями. Теперь можно пе-

рейти к самому интересному, а именно, – к расширению класса ImageField для создания миниатюр.

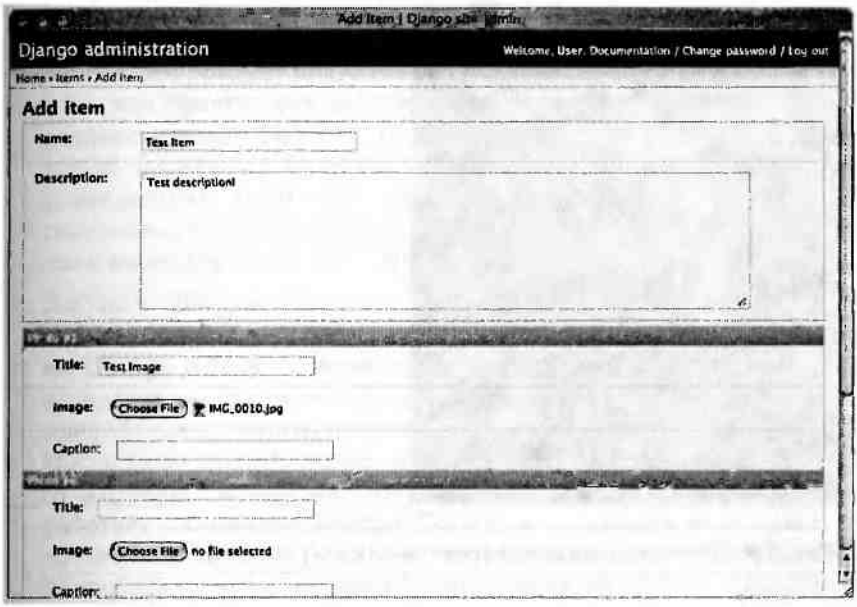

Рис. 7.2. ImageField

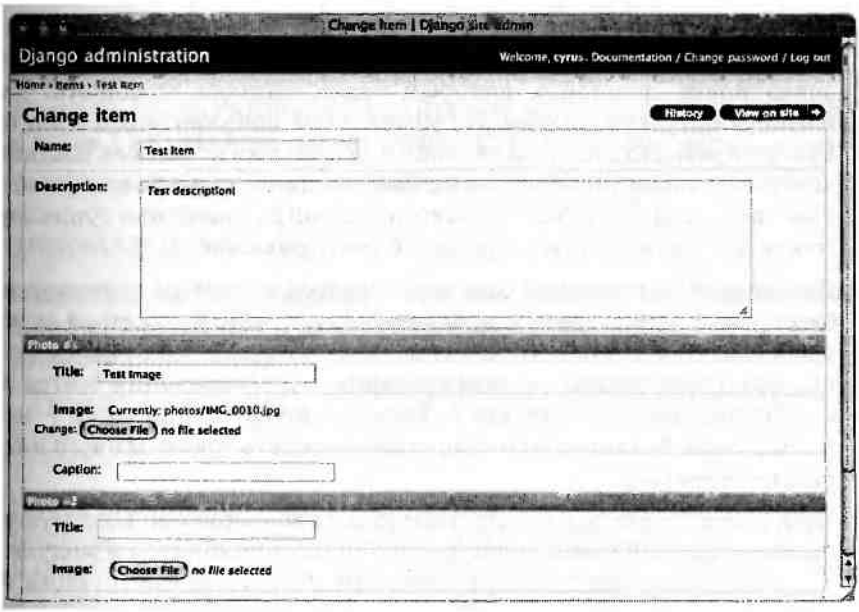

Рис. 7.3. Административный интерфейс после выгрузки файла

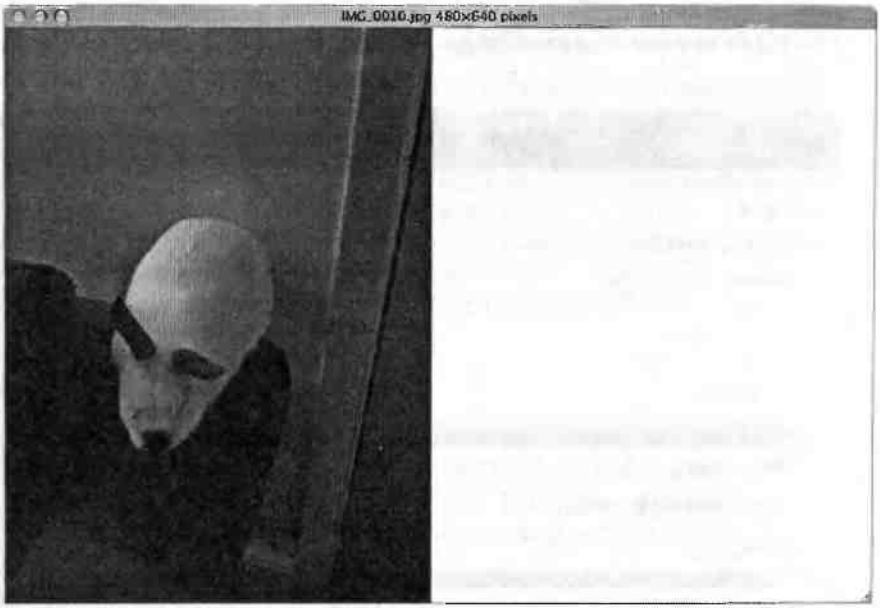

Рис. 7.4. Проверка текущего значения поля ImageField

Создание нашего собственного поля файла

Поскольку платформа Django не имеет встроенных средств создания миниатюр, мы создадим свой собственный класс поля модели, унаследовав класс `ImageField`, который будет способен создавать, удалять и отображать миниатюры. В официальной документации к платформе Django превосходно представлена информация о том, как создавать совершенно новые поля моделей с чистого листа, однако в данном случае нам требуется всего лишь немного изменить поведение существующего класса, что несколько проще и более привычно.

Для этого нам потребовалось переопределить четыре метода родительского класса `ImageField` и добавить один простой, частный метод для удобства. Исходный программный код, который лег в основу этой главы, содержит обширные комментарии – документация всегда оказывает существенную помощь в освоении новых территорий – но здесь мы удалили большую их часть, чтобы сделать программный код более удобочитаемым.

Наш класс `ImageField` с поддержкой воспроизведения миниатюр, естественно, получил имя `ThumbnailImageField` и находится в модуле `gallery.items.fields`. Этот модуль содержит несколько инструкций импортирования, вспомогательную функцию и пару подклассов, расширяющих возможности некоторых встроенных классов Django. Если вы не

Не бойтесь исходных текстов

Программисты обычно воспринимают библиотеки, с которыми им приходится работать, – даже распространяемые с открытыми исходными текстами – как «черные ящики», про которые известно, что следует подавать на вход и что получается на выходе, а то, что происходит внутри, окутано тайной. Хотя это и справедливо в некоторых случаях – например, когда библиотеки пишутся на языках низкого уровня или на языках со сложным синтаксисом, из-за чего библиотеки получаются поистине гигантскими или запутанными, – но к исходному программному коду на языке Python это обычно не относится.

Хорошо продуманные следующие принципам «питонизма» библиотеки обычно легко доступны для понимания, и платформа Django в этом отношении не является исключением. Мы не утверждаем, что весь программный код платформы идеально проработан и откомментирован, но в значительной степени это так; и даже разработчики средней руки получат немалую пользу от изучения исходных текстов, если захотят разобраться, как работает платформа Django. То, что мы делаем в этой главе, недостаточно полно отражено в документации, но все необходимые подробности относительно легко можно выяснить, ознакомившись с реализацией класса `django.db.models.ImageField` и его родительских классов.

знакомы с особенностями объектно-ориентированного программирования на языке Python, обращайтесь за дополнительной информацией к главе 1 «Практическое введение в Python для Django».

Теперь пройдемся по этому файлу сверху вниз.

Инициализация

Любой файл с программным кодом на языке Python, за редким исключением, начинается с инструкций импортирования, и этот файл – не исключение.

```
from django.db.models.fields.files import ImageField, ImageFieldFile
from PIL import Image
import os
```

В этих инструкциях нет ничего сложного: все, что нам необходимо для решения поставленной задачи, – это родительские классы `ImageField` и `ImageFieldFile`, класс `Image` из библиотеки PIL, который обеспечит создание миниатюр, и встроенный модуль `os` для работы с файлами миниатюр.

8 Зак. 1496

```
def _add_thumb(s):
    """
    Изменяет строку (имя файла, URL), содержащую имя файла изображения,
    вставляя '.thumb' перед расширением имени файла
    (которое изменяется на '.jpg').
    """
    parts = s.split(".")
    parts.insert(-1, "thumb")
    if parts[-1].lower() not in ['jpeg', 'jpg']:
        parts[-1] = 'jpg'
    return ".".join(parts)
```

Как следует из строки документирования (всегда старайтесь добавлять содержательные строки документирования), вспомогательная функция _add_thumb принимает путь к оригинальному файлу изображения и вставляет в него строку «.thumb». Таким образом, для выгруженного файла изображения с именем rabbit.jpg будет получено имя файла с миниатюрой rabbit.thumb.jpg, а поскольку наш программный код может создавать миниатюры изображений только в формате JPEG, функция изменяет еще и расширение имени файла, если это необходимо.

```
class ThumbnailImageField(ImageField):
    """
    Ведет себя так же, как обычный класс ImageField, но дополнительно
    сохраняет миниатюру (JPEG) изображения и предоставляет методы
    get_FIELD_thumb_url() и get_FIELD_thumb_filename().

    Принимает два дополнительных, необязательных аргумента: thumb_width
    и thumb_height, каждый из которых имеет значение по умолчанию
    128 (пикселей). При изменении размеров отношение ширины к высоте
    сохраняется, обеспечивая пропорциональность изображения;
    за дополнительной информацией обращайтесь к описанию метода
    Image.thumbnail() в библиотеке PIL.
    """
    attr_class = ThumbnailImageFieldFile
```

Здесь мало что можно сказать – мы определяем новый подкласс, породив его от класса ImageField и добавив большую и замечательную строку документирования. Благодаря ей любой, кто воспользуется справочной системой языка Python или автоматизированными инструментами получения документации, сможет получить представление о том, что делает наш программный код.

Единственная здесь строка программного кода, относящаяся к атрибуту attr_class, изменяет специальный класс, используемый нашим полем в качестве делегата для доступа к атрибуту. Подробнее об этом классе мы поговорим в следующем разделе. Последняя часть этого введения – метод __init__:

```
def __init__(self, thumb_width=128, thumb_height=128, *args, **kwargs):
    self.thumb_width = thumb_width
```

```
        self.thumb_height = thumb_height
        super(ThumbnailImageField, self).__init__(*args, **kwargs)
```

Наша реализация метода __init__ также не содержит ничего сложного – мы просто сохраняем желательную максимальную ширину и высоту миниатюры для использования во время выполнения операции изменения размера. Это упростит использование поля с миниатюрами различных размеров.

Добавление атрибутов в поле

В большинстве своем поля весьма консервативны и не изменяют содержащие их объекты моделей, но в нашем случае нам необходимо обеспечить простой доступ к дополнительной информации, которую мы предоставляем (имена файлов и адреса URL миниатюр). Решение состоит в том, чтобы создать подкласс специального класса ImageFieldFile, используемого полем ImageField для управления своими атрибутами, например при поиске атрибутов самого поля.

Например, чтобы получить путь к файлу с изображением для поля image типа ImageField, вы обращаетесь к myobject.image.path, где в данном случае path – это атрибут объекта ImageFieldFile. Платформа Django кэширует данные в файлах, когда это возможно, и делегирует операции с файлами на более низкий уровень, поэтому данная особенность реализована с помощью свойств. (Посмотрите главу 1, чтобы освежить в памяти сведения о встроенной функции property.)

Следующий фрагмент иллюстрирует, как реализовано свойство ImageFieldFile.path в исходных текстах Django:

```
    def _get_path(self):
        self._require_file()
        return self.storage.path(self.name)
    path = property(_get_path)
```

Этот фрагмент взят из реализации класса FieldFile (предок класса ImageFieldFile, который в свою очередь, как вы уже догадались, используется классом ImageField). Вспомните предыдущую вспомогательную функцию _add_thumb и как она преобразовывала заданный путь к файлу, – и вы сможете догадаться, что нужно сделать, чтобы добавить атрибуты thumb_path и thumb_url к нашему полю:

```
class ThumbnailImageFieldFile(ImageFieldFile):
    def _get_thumb_path(self):
        return _add_thumb(self.path)
    thumb_path = property(_get_thumb_path)

    def _get_thumb_url(self):
        return _add_thumb(self.url)
    thumb_url = property(_get_thumb_url)
```

Поскольку методы чтения path и url уже определены и для обеспечения безопасности операций требуется совсем немного шаблонного программного кода (вызов self._require_file, который мы видели в предыдущем фрагменте, обращавшемся к _get_path), мы опустим весь дополнительный программный код. Здесь мы просто выполняем преобразование _add_thumb и присваиваем полученный результат атрибуту с требуемым именем с помощью функции property.

Благодаря такому определению класса ThumbnailImageFieldFile, расположенному перед определением класса ThumbnailImageField, и ссылке attr_class, расположенной в начале определения класса ThumbnailImageField, у нашего поля появилось два новых атрибута, которые можно задействовать в программном коде на языке Python или в шаблонах: myobject.image.thumb_path и myobject.image.thumb_url (помня, конечно, что myobject – это экземпляр модели Django, а image – это поле ThumbnailImageField в модели).

Создание подкласса ImageFieldFile и необходимость связывания этого подкласса с подклассом ImageField – вероятно, не самое очевидное действие, которое необходимо выполнить; в большинстве случаев при создании своих полей моделей не требуется заходить так далеко. В действительности вам, как пользователю Django, скорее всего, никогда не придется сталкиваться с этим аспектом моделей (хотя такая возможность стала более доступной, чем раньше, – предыдущая версия этого раздела была несколько сложнее). Однако этот пример наглядно показывает, что команда разработчиков Django стремится обеспечить расширяемость не только со стороны общедоступного API платформы, но и для внутренних структур.

Теперь, когда у нас имеется возможность получить адрес URL миниатюры и путь к файлу в файловой системе, можно перейти к реализации создания (и удаления) файла миниатюры.

Сохранение и удаление миниатюры

По сути дела для реализации операции создания самого файла миниатюры необходимо переопределить метод save в классе ThumbnailImageFieldFile (не в классе ThumbnailImageField!), как показано ниже:

```
def save(self, name, content, save=True):
    super(ThumbnailImageFieldFile, self).save(name, content, save)
    img = Image.open(self.path)
    img.thumbnail(
        (self.field.thumb_width, self.field.thumb_height),
        Image.ANTIALIAS
    )
    img.save(self.thumb_path, 'JPEG')
```

Вызов метода save суперкласса выполняет обычную операцию сохранения основного файла изображения, поэтому дальнейшая работа на

шего метода заключается в выполнении последовательности из трех действий: открыть основной файл изображения, создать миниатюру и сохранить миниатюру в файле с именем миниатюры. Обратите внимание, что имя self.field обеспечивает доступ к полю Field, которому принадлежит данный объект File и где хранятся желаемые размеры миниатюры. Использование класса Image из библиотеки PIL, импортированного в самом начале, обеспечивает максимальную простоту нашему программному коду.

Наконец, нам необходимо обеспечить удаление миниатюр при удалении «родительских» изображений:

```
def delete(self, save=True):
    if os.path.exists(self.thumb_path):
        os.remove(self.thumb_path)
    super(ThumbnailImageFieldFile, self).delete(save)
```

Благодаря выразительности языка Python этот фрагмент сам описывает себя. Сначала мы получаем имя файла миниатюры, удаляем его (если он существует, конечно – нет смысла возбуждать исключение, если файл отсутствует) и сообщаем суперклассу, что он может выполнить свои действия по удалению файла (в результате которых удаляется оригинальный файл изображения). Обратите внимание: метод delete, как и метод save, вызывается объектом ImageField при удалении вмещающего объекта модели.

Порядок выполнения операций

Порядок выполнения операций в нашем методе delete имеет большое значение и его всегда следует учитывать при создании своих подклассов. Если бы мы вызвали функцию super первой, то вызов метода self.thumb_path мог бы завершиться с ошибкой, потому что он, в свою очередь, вызывает метод self.path, который, если вспомнить предыдущий фрагмент, требует, чтобы основной файл существовал! Поэтому, чтобы избежать появления ошибки, нам необходимо выполнить удаление этого файла как можно позже.

Использование ThumbnailImageField

Теперь, когда мы определили подкласс класса ImageField, его можно задействовать в работе. Все, что для этого нужно, – это добавить новую инструкцию импортирования в файл models.py:

```
from gallery.items.fields import ThumbnailImageField
```

и заменить `models.ImageField` на нашу версию `ThumbnailImageField` в модели `Photo`:

```
class Photo(models.Model):
    item = models.ForeignKey(Item)
    title = models.CharField(max_length=100)
    image = ThumbnailImageField(upload_to='photos')
    caption = models.CharField(max_length=250, blank=True)
```

После перезапуска веб-сервера мы не увидим существенных изменений в административной части сайта, как показано на рис. 7.5, потому что мы не изменили ничего, что имело бы отношение к внешнему виду поля.

После выгрузки файла все выглядит точно так же, как и раньше, в чем можно убедиться, взглянув на рис. 7.6. Однако, заглянув в каталог с выгруженными файлами, мы увидим плоды нашего труда.

```
user@example:/var/www/gallery/media/photos $ ls -l
total 148
-rw-r-r- 1 www-data www-data 140910 2008-03-30 22:15 IMG_0010.jpg
-rw-r-r- 1 www-data www-data   1823 2008-03-30 22:15 IMG_0010.thumb.jpg
```

Получилось! Правда, нам еще придется приложить некоторые усилия, чтобы добраться до просмотра миниатюры, так как для этого необходимо рассмотреть шаблоны нашего приложения, отображающие эти

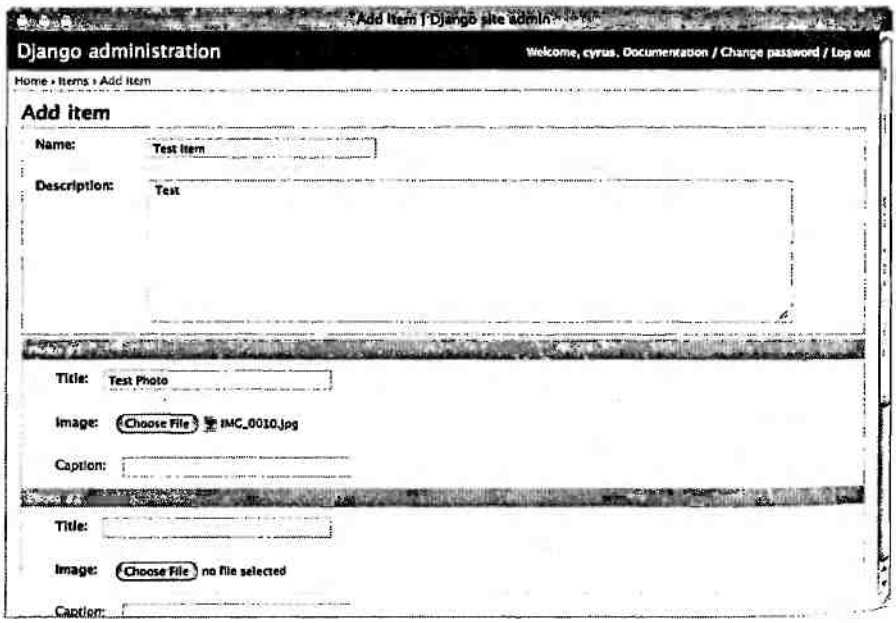

Рис. 7.5. Никаких отличий от того, что было раньше

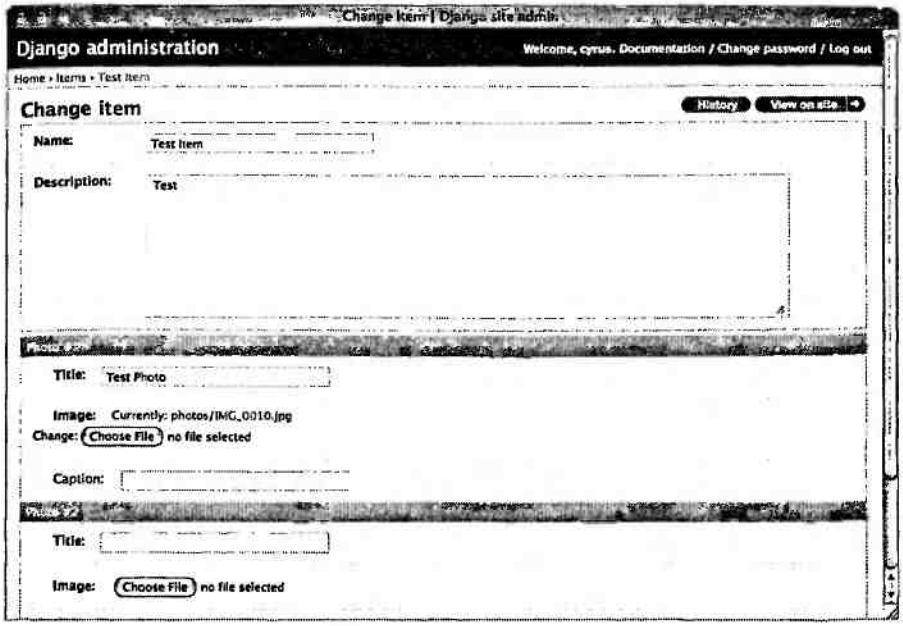

Рис. 7.6. В визуальном отображении по-прежнему нет ничего нового

миниатюры. Но прежде чем перейти к ним, коротко рассмотрим еще одну сторону нашего приложения: как применить принцип «не повторяйся» к набору адресов URL.

Применение принципа «не повторяйся» к адресам URL

До сих пор все свое внимание мы уделяли модели приложения галереи. Настало время пройтись по структуре адресов URL, чтобы подготовить почву к исследованию шаблонов в следующем разделе. Однако сначала мы объясним, в чем заключается необычность организации адресов. Это обусловлено целями, преследуемыми при разработке приложения, – нам требовалось, чтобы приложение одинаково хорошо работало как на верхнем уровне домена (например, *http://www.example.com/*), так и в каком-либо из подразделов (например, *http://www.example.com/gallery/*).

По умолчанию предполагается, что сайт на платформе Django работает в первой ситуации – адреса URL анализируются от корня домена, даже если обработчик веб-сервера прикреплен выше.

Вследствие этого адреса URL должны включать в себя весь путь URL, то есть все адреса сайта `/gallery/` должны начинаться с этой под-

строки.[1] Мы просто сделали очевидную вещь – сохранили значени в виде переменной в файле settings.py и использовали ее везде, где это необходимо.

```
ROOT_URL = '/gallery/'
```

Поскольку в файле settings.py существует еще пара переменных, использующих пути URL, то мы использовали эту переменную для определения URL страницы регистрации, URL папки с изображениями и URL папки с изображениями для приложения администрирования.

```
LOGIN_URL = ROOT_URL + 'login/'
MEDIA_URL = ROOT_URL + 'media/'
ADMIN_MEDIA_PREFIX = MEDIA_URL + 'admin/'
```

Затем, вследствие особенностей работы системы включения адресов URL в платформе Django, мы должны провести настройку двух корневых файлов URLconf, где «нормальный» файл urls.py верхнего уровня просто использует переменную ROOT_URL и вызывает «настоящий» файл urls.py, который ничего не знает о переменной ROOT_URL и ее назначении. Ниже приводится содержимое корневого файла urls.py:

```
from django.conf.urls.defaults import *
from gallery.settings import ROOT_URL

urlpatterns = patterns('',
    url(r'^%s' % ROOT_URL[1:], include('gallery.real_urls')),
)
```

Примечание

Нам пришлось применить к переменной ROOT_URL операцию извлечения среза, чтобы отсечь начальный символ слеша, потому что при определении переменных в файле settings.py, таких как LOGIN_URL, начальный символ слеша необходим, чтобы обеспечить корректность адресов URL. Однако, из-за того что механизм анализа адресов URL платформы Django не учитывает начальный символ слеша, нам следует избавиться от него, чтобы анализ адресов URL выполнялся без ошибок.

Ниже приводится содержимое «настоящего» корневого файла URLconf, которому без лишних фантазий мы дали имя real_urls.py:

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

urlpatterns = patterns('',
```

[1] В Django 1.0 появилась новая директива конфигурации веб-сервера Apache PythonOption django.root <root>, которая во многом замещает параметр настройки ROOT_URL, рассматриваемый здесь. Однако мы решили оставить эту часть главы нетронутой как пример того, что платформа Django – это «всего лишь Python», и она позволяет добиться одного и того же разными способами.

```
        url(r'^admin/(.*)', admin.site.root),
        url(r'^', include('gallery.items.urls')),
    )
```

Наконец, в шаблонах удобно иметь доступ к ROOT_URL, чтобы конструировать похожие включения адресов URL, когда потребуется подключать файлы CSS или JavaScript. Этого можно добиться с помощью простого процессора контекста (о которых рассказывалось в главе 6, «Шаблоны и обработка форм»).

```
from gallery.settings import ROOT_URL

def root_url_processor(request):
    return {'ROOT_URL': ROOT_URL}
```

Вот и все! После внесения этих изменений в обычный проект Django все будет зависеть от переменной ROOT_URL, которая в настоящий момент имеет значение '/gallery/', то есть, как упоминалось выше, приложение должно быть доступно по адресу *http://www.example.com/gallery/*. Если вдруг потребуется развернуть приложение по адресу *http://www.example.com/*, достаточно будет изменить значение переменной ROOT_URL на '/' (и изменить конфигурацию веб-сервера, чтобы платформа Django оказалась подключенной к корневому уровню).

Схема адресов элементов Item приложения

Завершая создание структуры адресов URL с применением принципа «не повторяйся», мы учтем три особенности реализации методов get_absolute_url в наших объектах. Первая и сама важная часть, с которой вы уже сталкивались в этой книге, – это использование функции url в определениях адресов в файлах URLconf, которая позволяет нам присваивать адресам уникальные имена. Ниже приводится содержимое файла urls.py в самом приложении items:

```
from django.conf.urls.defaults import *
from gallery.items.models import Item, Photo

urlpatterns = patterns('django.views.generic',
    url(r'^$', 'simple.direct_to_template',
        kwargs={
            'template': 'index.html',
            'extra_context': {'item_list': lambda: Item.objects.all()}
        },
        name='index'
    ),
    url(r'^items/$', 'list_detail.object_list',
        kwargs={
            'queryset': Item.objects.all(),
            'template_name': 'items_list.html',
            'allow_empty': True
        },
```

```
        name='item_list'
    ),
    url(r'^items/(?P<object_id>\d+)/$', 'list_detail.object_detail',
        kwargs={
            'queryset': Item.objects.all(),
            'template_name': 'items_detail.html'
        },
        name='item_detail'
    ),
    url(r'^photos/(?P<object_id>\d+)/$', 'list_detail.object_detail',
        kwargs={
            'queryset': Photo.objects.all(),
            'template_name': 'photos_detail.html'
        },
        name='photo_detail'
    ),
)
```

Как видите, приложение содержит главную страницу, страницу со списком фотоальбомов, по странице для каждого фотоальбома и по странице для каждой фотографии. Каждому адресу URL присвоено собственное очевидное имя. Эти имена могут использоваться в шаблонах, в виде тега {% url %}, как будет показано в следующем разделе, а также в декораторе permalink, обертывающем метод get_absolute_url, как показано ниже:

```
class Item(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()

    class Meta:
        ordering = ['name']

    def __unicode__(self):
        return self.name

    @permalink
    def get_absolute_url(self):
        return ('item_detail', None, {'object_id': self.id})
```

Декоратор permalink ожидает, что обертываемая им функция будет возвращать кортеж, состоящий из трех элементов: имени URL, списка позиционных аргументов и словаря именованных аргументов, используемых для воссоздания URL. Как видно из предыдущего примера, функция представления одного фотоальбома не принимает позиционные аргументы и принимает один именованный аргумент, который и возвращается нашим методом get_absolute_url.

С такими настройками метод Item.get_absolute_url будет возвращать соответствующий адрес URL даже в случае изменения структуры адресов URL, обеспечивая тем самым следование принципу «не повто-

ряйся» (правда при этом, если убрать декоратор, метод get_absolute_url будет вести себя довольно странно).

Соединяем все это с шаблонами

Наконец, после создания собственного поля модели и настройки структуры адресов URL все, что осталось сделать, благодаря применению универсальных представлений, – это реализовать шаблоны. С целью соблюдения принципа «не повторяйся» мы будем использовать прием наследования и начнем с шаблона, составляющего основу структуры, – содержащего простую таблицу CSS.

```
<html>
    <head>
        <title>Gallery - {% block title %}{% endblock %}</title>
        <style type="text/css">
            body { margin: 30px; font-family: sans-serif; background: #fff; }
            h1 { background: #ccf; padding: 20px; }
            h2 { background: #ddf; padding: 10px 20px; }
            h3 { background: #eef; padding: 5px 20px; }
            table { width: 100%; }
            table th { text-align: left; }
        </style>
    </head>
    <body>
        <h1>Gallery</h1>
        {% block content %}{% endblock %}
    </body>
</html>
```

Далее следует шаблон главной страницы. В приложении, послужившем прототипом для этой главы, главная страница содержала дополнительную функциональность, напоминающую функциональность системы управления содержимым, которая позволяла редактировать текст приветствия со стороны административного раздела сайта; мы опустили ее здесь, чтобы сделать пример более коротким. Вместо этого мы используем статический текст и выводим короткий список из трех выделенных фотоальбомов Item, адреса которых определяются в файле URLconf. (В настоящее время этот список содержит все имеющиеся фотоальбомы, но логику работы шаблона легко можно изменить, задействовав некоторый другой критерий.)

```
{% extends "base.html" %}

{% block title %}Home{% endblock %}
{% block content %}

<h2>Welcome to the Gallery!</h3>
<p>Here you find pictures of various items. Below are some highlighted
items; use the link at the bottom to see the full listing.</p>
```

```
<h3>Showcase</h3>
<table>
    <tr>
    {% for item in item_list|slice:":3" %}
        <td>
            <a href="{{ item.get_absolute_url }}"><b>{{ item.name }}</b><br />
             {% if item.photo_set.count %}
                 <img src="{{ item.photo_set.all.0.image.thumb_url }}" />
             {% else %}
                 <span>No photos (yet)</span>
             {% endif %}
             </a>
        </td>
    {% endfor %}
    </tr>
</table>
<p><a href="{% url item_list %}">View the full list &raquo;</a></p>

{% endblock %}
```

Этот шаблон воспроизводит страницу, как показано на рис. 7.7.

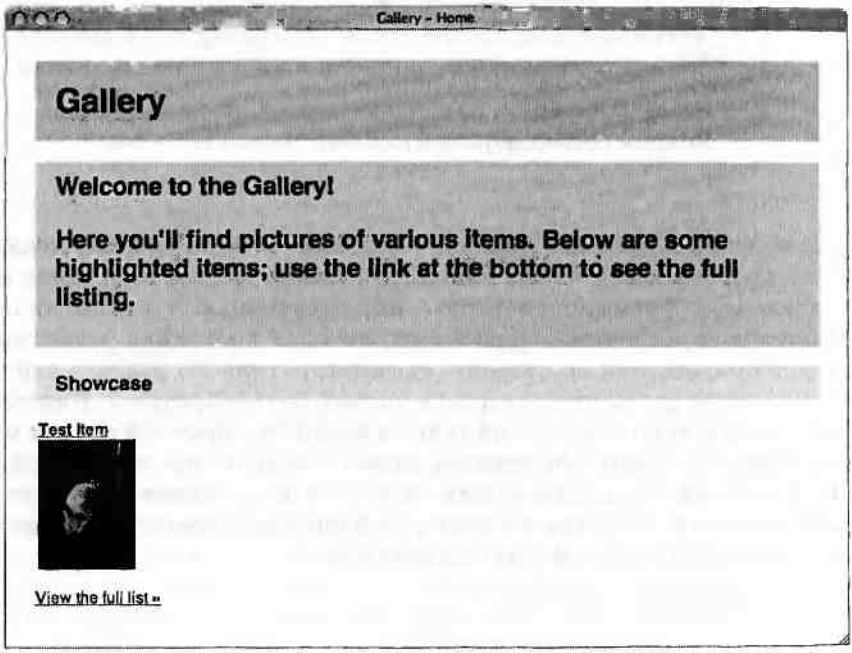

Рис. 7.7. Главная страница приложения gallery

Обратите внимание на использование метода get_absolute_url и тега {% url %} для создания ссылок на страницу фотоальбома и на страницу со списком фотоальбомов, соответственно; и – особенно – на использование метода image.thumb_url первого изображения из каждого списка фотографий Item. Для более гибкого решения вопроса о том, какая миниатюра будет представлять фотоальбом, можно было бы добавить в модель Photo признак, определяющий «представительность» изображения; и это лишь одно из многих направлений возможного расширения приложения.

Страница со списком фотоальбомов (items_listing.html) – это лишь более полная версия списка выделенных фотоальбомов на главной странице, и здесь используются те же приемы. Внешний вид страницы приводится на рис. 7.8.

```
{% extends "base.html" %}

{% block title %}Item List{% endblock %}

{% block content %}
<p><a href="{% url index %}">&laquo; Back to main page</a></p>

<h2>Items</h2>
{% if object_list %}
<table>
    <tr>
        <th>Name</th>
        <th>Sample Thumb</th>
        <th>Description</th>
    </tr>
    {% for item in object_list %}
    <tr>
        <td><i>{{ item.name }}</i></td>
        <td>
            {% if item.photo_set.count %}
             <a href="{{ item.get_absolute_url }}">
                <img src="{{ item.photo_set.all.0.image.thumb_url }}" />
            </a>
            {% else %}
             (No photos currently uploaded)
            {% endif %}
        </td>
        <td>{{ item.description }}</td>
    </tr>
    {% endfor %}
</table>
{% else %}
<p>There are currently no items to display.</p>
{% endif %}

{% endblock %}
```

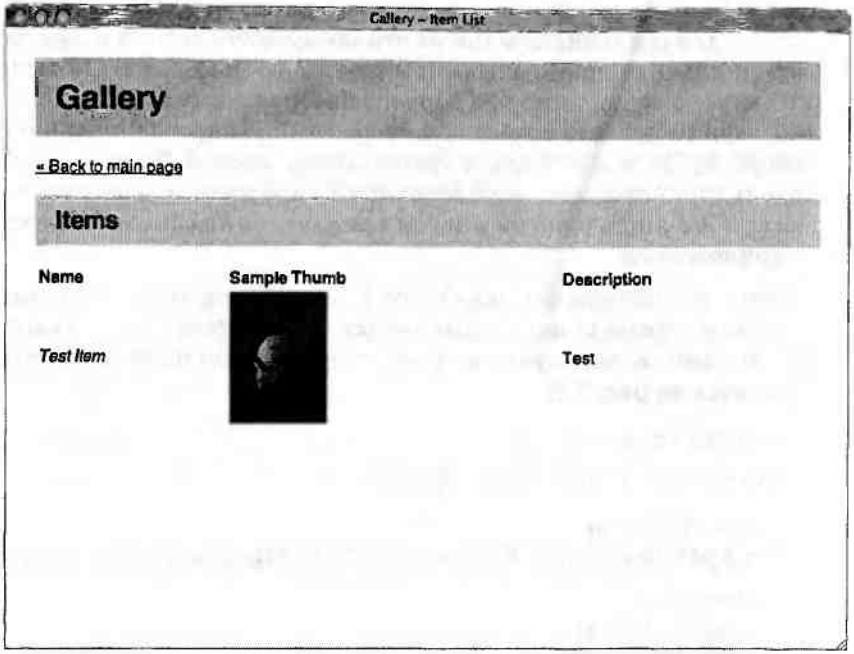

Рис. 7.8. Страница со списком фотоальбомов в галерее

Точно так же страница представления одного фотоальбома (`items_detail.html`) напоминает страницу со списком фотоальбомов – за исключением того, что в ней выводится список всех фотографий вместо одной фотографии-представителя, что показано на рис. 7.9.

```
{% extends "base.html" %}

{% block title %}{{ object.name }}{% endblock %}

{% block content %}

<p><a href="{% url item_list %}">&laquo; Back to full listing</a></p>
<h2>{{ object.name }}</h2>
<p>{{ object.description }}</p>

<h3>Photos</h3>
<table>
    <tr>
        <th>Title</th>
        <th>Thumbnail</th>
        <th>Caption</th>
    </tr>
    {% for photo in object.photo_set.all %}
    <tr>
        <td><i>{{ photo.title }}</i></td>
```

```
        <td>
            <a href="{{ photo.get_absolute_url }}">
                <img src="{{ photo.get_image_thumb_url }}" />
            </a>
        </td>
        <td>{{ photo.caption }}</td>
    </tr>
    {% endfor %}
</table>

{% endblock %}
```

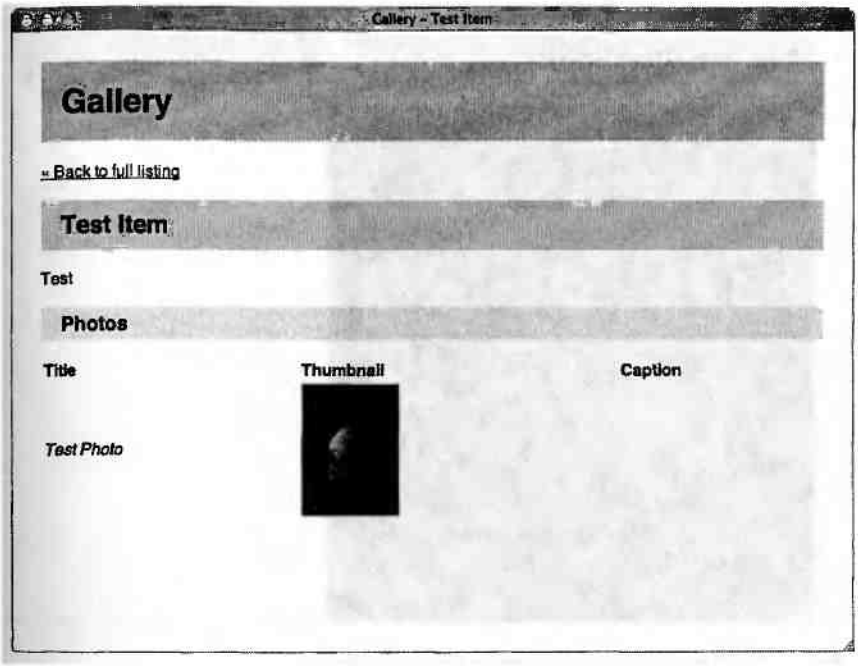

Рис. 7.9. Страница просмотра фотоальбома

Наконец, ниже приводится шаблон страницы просмотра одной фотографии (photos_detail.html), – единственное место приложения, где используется атрибут image.url. Внешний вид страницы приводится на рис. 7.10.

```
{% extends "base.html" %}

{% block title %}{{ object.item.name }} - {{ object.title }}{% endblock %}

{% block content %}

<a href="{{ object.item.get_absolute_url }}">&laquo; Back to {{
object.item.name }} detail page</a>
```

```
<h2>{{ object.item.name }} - {{ object.title }}</h2>
<img src="{{ object.get_image_url }}" />
{% if object.caption %}<p>{{ object.caption }}</p>{% endif %}

{% endblock %}
```

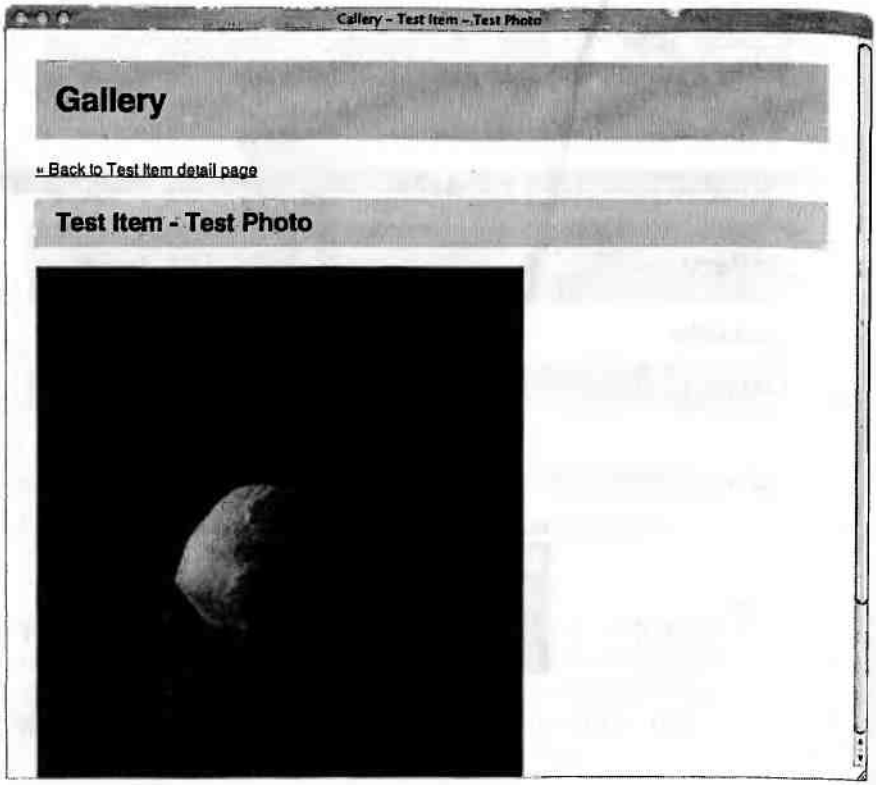

Рис. 7.10. Страница просмотра фотографии

В заключение

Это было что-то вроде короткой экскурсии, но – надеемся – к этому моменту у вас сложилось достаточно полное представление о том, как соединяются различные компоненты приложения.

- Мы определили модели и использовали приложение администрирования, чтобы продемонстрировать, как работает механизм выгрузки изображений, включая все необходимые подготовительные настройки системы.
- Задача вывода миниатюр заставила нас определить новые подклассы класса поля изображения и связанного с ним класса файла, в ко-

торых мы просто переопределили несколько методов, чтобы получить возможность изменять размеры миниатюр и обеспечить доступ к файлам миниатюр.

- Мы использовали все возможности определения схемы адресов URL, чтобы обеспечить следование принципу «не повторяйся», включая реализацию параметра настройки «корневого URL» (похожего на тот, что был добавлен в ядро Django непосредственно перед выходом версии 1.0), позволившую нам обеспечить гибкую поддержку адресов URL.
- Наконец, мы создали простые шаблоны, которые дают пользователям возможность перемещаться по сайту и просматривать фотографии.

8

Система управления содержимым

Начинающие пользователи Django часто спрашивают: «Существует ли открытая система управления содержимым (CMS – Content Management System), написанная на платформе Django?». Наш типичный ответ обычно не тот, какой они хотели бы услышать: вы сами можете создать свою систему. В этой главе рассматриваются два способа, как это можно сделать. Первый заключается в использовании вспомогательного приложения, упрощающего создание и публикацию «плоских» страниц HTML, с последующим расширением его возможностей до простой системы управления содержимым.

Что такое система управления содержимым?

Люди по-разному представляют себе систему управления содержимым. Возможно, проще создать свою систему на платформе Django, чем пытаться адаптировать чье-то решение, за исключением случая, когда это решение – то, что вам требовалось.

Ярлык «Система управления содержимым» используется применительно к приложениям самых разных типов. Для кого-то это означает простой интерфейс редактирования содержимого веб-страниц и отображение его в такие шаблоны, как блог. А для кого-то это понятие может включать в себя систему прав доступа и правил документооборота, возможность создания выходных документов в разных форматах из одного источника, способность хранить различные версии и редакции любых видов содержимого (а также индексирование, архивирование и управление содержимым).

Другими словами, практически каждая система управления содержимым до определенной степени является специализированной. А единственная цель Django состоит в том, чтобы упростить процесс разра-

ботки специализированных веб-приложений. Хорошо ориентирующиеся в реалиях разработчики справедливо считают, что попытка создать свою систему управления содержимым сродни повторному изобретению колеса, но кто знает – может быть, ваше конкретное колесо еще не было изобретено.

Экосистема открытых приложений на платформе Django продолжает развиваться, и вам наверняка встретятся приложения, относящиеся к категории систем управления содержимым, достаточно зрелые, чтобы их сопровождали интенсивные рекомендации к использованию и окружали свои сообщества пользователей и разработчиков. Система, прекрасно отвечающая вашим потребностям, возможно, уже есть среди этих разработок. Поэтому оглядитесь по сторонам (некоторые указания вы найдете в приложении D, «Поиск, оценка и использование приложений на платформе Django»), хотя и не забрасывайте идею реализовать свое собственное решение.

Альтернатива системе управления содержимым: Flatpages

Для создания простейшей системы управления содержимым на платформе Django вообще не потребуется написать ни строчки программного кода. В составе Django распространяется приложение с названием «Flatpages», которое отлично подходит для простых случаев. Самое привлекательное, что для запуска этого приложения потребуется выполнить совсем немного настроек и не придется писать программный код, который требовалось бы сопровождать.

Другое удобство состоит в том, что при использовании приложения Flatpages адреса URL веб-страниц задаются с помощью приложения администрирования – вам не придется редактировать файл URLconf, чтобы добавить новую страницу. Однако, прежде чем вы придете в восхищение, взгляните на некоторые ограничения этого приложения.

- Все страницы могут редактироваться всеми пользователями, обладающими правом административного доступа к приложению Flatpages, – пользователи не могут «владеть» отдельными страницами.
- Кроме атрибутов `title` и `content`, а также нескольких полей специализированного назначения, которые будут рассматриваться ниже, объект Flatpages не обладает никакими полезными возможностями – он не поддерживает ни дату создания, ни какие-либо другие данные, которые можно было бы ассоциировать с конкретными страницами.
- Поскольку это приложение предоставляется платформой, вы не сможете легко и просто изменить его параметры администрирования, добавить новые поля или методы в модели.

Если эти ограничения не кажутся вам существенными, то приложение Flatpages может оказаться для вас полезным. Мы покажем, как настраивать и использовать Flatpages в следующих нескольких разделах, а затем посмотрим, как создать более надежное приложение системы управления содержимым.

Включение приложения Flatpages

Ниже приводится краткий перечень последовательности действий, которые необходимо выполнить, чтобы настроить и запустить Flatpages.

1. Создать новый проект Django с помощью утилиты `django-admin.py`.
2. Открыть файл проекта `settings.py` и изменить параметр `MIDDLEWARE_CLASSES`, добавив в него `django.contrib.flatpages.middleware.FlatpageFallbackMiddleware`.
3. Добавить приложения `django.contrib.flatpages` и `django.contrib.admin` в параметр `INSTALLED_APPS` в файле `settings.py`.
4. Запустить команду `manage.py syncdb`, чтобы создать все необходимые таблицы.
5. Отредактировать файл `urls.py`, раскомментировав строку с настройками URL приложения администрирования.
6. (Пере)запустить свой веб-сервер.

Как только вы выполните все описанные действия и откроете страницу администратора сайта, вы должны увидеть нечто аналогичное рис. 8.1.

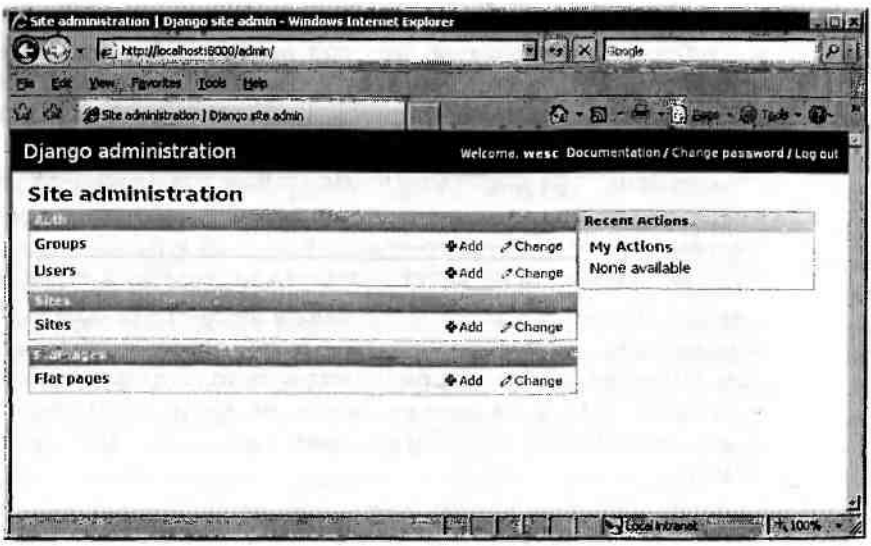

Рис. 8.1. Страница администратора после регистрации при активированном приложении Flatpages

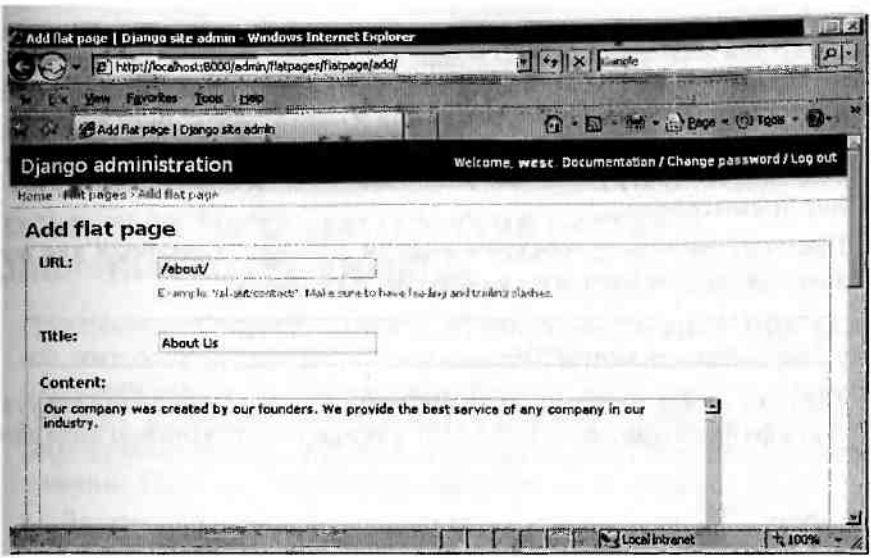

Рис. 8.2. Форма добавления страницы в приложении Flatpage

Щелкните на кнопке Add (Добавить), чтобы добавить новый объект Flatpage (рис. 8.2). Обязательными являются только поля URL, Title (Заголовок) и Content (Содержимое). Обратите внимание, что адрес URL должен начинаться и заканчиваться символом слеша, как показано на рисунке.

Создайте одну-две страницы, чтобы было что использовать, когда вы дойдете до этапа тестирования!

В следующем разделе мы сразу же перейдем к шаблонам, потому что приложение Flatpages не требует дополнительной настройки адресов URL кроме той, что мы уже выполнили. Вместо этого оно использует механизм промежуточной обработки, который перехватывает ошибку 404 и пытается отыскать требуемый URL в списке объектов Flatpage. Если соответствующий объект обнаруживается, управление передается приложению Flatpage. В противном случае производится обычная обработка ошибки 404.

Примечание

Прием перехвата ошибки 404 в приложении Flatpage означает, что оно также может использоваться обычными приложениями на платформе Django, что позволит легко определить свои статические страницы («О нас» или «Официальная информация») без необходимости постоянно обновлять файлы URLconf.

Шаблоны Flatpages

Отдельные объекты Flatpage имеют атрибут `template_name`, значение которого доступно для изменения, но по умолчанию приложение Flat-

pages пытается отыскать шаблон flatpages/default.html. Из этого следует, что вам необходимо создать каталог для шаблонов с именем «flatpages» в одном из местоположений, перечисленных в параметре настройки TEMPLATE_DIRS проекта или в каталоге «templates» внутри одного из приложений, перечисленных в параметре INSTALLED_APPS, если используется загрузчик шаблонов app_directories. Создайте этот каталог прямо сейчас.

Шаблону передается объект с именем flatpage, доступный для использования вполне очевидным образом. Например:

```
<h1>{{ flatpage.title }}</h1>
<p>{{ flatpage.content }}</p>
```

Итак, создайте у себя простой шаблон страницы, сохранив следующий код в файле с именем default.html в только что созданном каталоге.

```
<html>
    <head>
        <title>My Dummy Site: {{ flatpage.title }}</title>
    </head>
    <body>
        <h1>{{ flatpage.title }}</h1>
        <p>{{ flatpage.content }}</p>
    </body>
</html>
```

Тестирование

Теперь попробуйте загрузить свою страницу. Например, если ваш сервер выполняется на вашей рабочей станции и вы создали в приложении администрирования объект Flatpage с адресом URL /about/, откройте в броузере страницу с адресом *http://localhost:8000/about/*. Приложение должно отобразить содержимое полей title и content с помощью шаблона default.html, как показано на рис. 8.3.

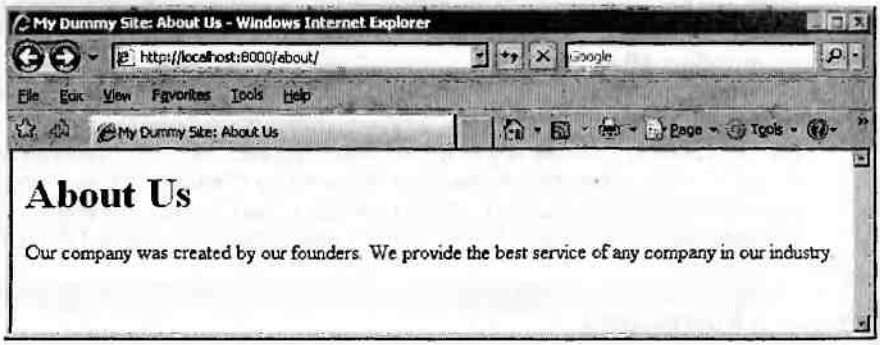

Рис. 8.3. Пример страницы «About Us» («О нас»)

Мы надеемся, что вы увидели достаточно, чтобы получить представление о том, как может использоваться приложение Flatpage и какую нишу оно может заполнить. А теперь перейдем к главному предмету обсуждения этой главы – к примеру более надежного приложения системы управления содержимым.

За рамками Flatpage: простая система управления содержимым

Приложение Flatpage, конечно, замечательно, но, как отмечалось выше, ему сопутствует ряд ограничений. Насколько легко или сложно они преодолимы, зависит от потребностей вашего сайта. Давайте рассмотрим процедуру использования платформы Django для создания собственной системы управления содержимым, не использующей приложение Flatpage. В частности, наше решение должно:

- Давать пользователям возможность вводить текст в удобном текстовом формате (не HTML), который затем автоматически будет преобразовываться в разметку HTML
- Создавать адреса URL страниц на основе текста, доступного для понимания человеком, а не на основе числовых значений первичных ключей базы данных
- Обеспечивать основные элементы производственного процесса – закреплять за каждой статьей определенного пользователя и позволять присваивать каждой статье признак принадлежности к одной из стадий подготовки
- Сохранять дату создания и последнего изменения
- Поддерживать классификацию статей с возможностью просмотра статей по категориям
- Предоставлять простую функцию поиска на каждой странице

Все эти возможности легко реализуются с помощью платформы Django. Большинство из них связано с особенностями, которые вы уже видели, – отчасти наука создания приложений на платформе Django заключается в изучении принципов эффективного комбинирования этих особенностей, чтобы получить конечный результат с наименьшими усилиями.

Для начала вам необходимо создать еще один проект (чтобы освежить в памяти процедуру создания проекта, его базы данных и т. д., обращайтесь к главе 2 «Django для нетерпеливых: создание блога»). Мы назвали свой проект `cmsproject` и единственное приложение в нем – `cms`.

Давайте начнем с модели.

Создание модели

Ниже приводится определение центральной модели нашей небольшой системы управления содержимым. Обратите внимание, что здесь присутствуют ссылки на две другие модели (User и Category), определения которых, а также несколько необходимых инструкций import мы скоро увидим.

```
class Story(models.Model):
    """Элемент информационного наполнения нашего сайта,
       обычно соответствует странице"""

    STATUS_CHOICES = (
        (1, "Needs Edit"),
        (2, "Needs Approval"),
        (3, "Published"),
        (4, "Archived"),
    )

    title = models.CharField(max_length=100)
    slug = models.SlugField()
    category = models.ForeignKey(Category)
    markdown_content = models.TextField()
    html_content = models.TextField(editable=False)
    owner = models.ForeignKey(User)
    status = models.IntegerField(choices=STATUS_CHOICES, default=1)
    created = models.DateTimeField(default=datetime.datetime.now)
    modified = models.DateTimeField(default=datetime.datetime.now)

    class Meta:
        ordering = ['modified']
        verbose_name_plural = "stories"

    @permalink
    def get_absolute_url(self):
        return ("cms-story", (), {'slug': self.slug})

class StoryAdmin(admin.ModelAdmin):
    list_display = ('title', 'owner', 'status', 'created', 'modified')
    search_fields = ('title', 'content')
    list_filter = ('status', 'owner', 'created', 'modified')
    prepopulated_fields = {'slug': ('title',)}

admin.site.register(Story, StoryAdmin)
```

Первый блок программного кода в объявлении класса модели определяет четыре стадии упрощенного представления производственного процесса. Конечно же, у вас процесс может включать и другие стадии.

Несмотря на то, что способ отображения вариантов для полей выбора, представленный здесь с помощью кортежа STATUS_CHOICES, обладает определенными преимуществами, тем не менее, в данном случае он привязан к числовым значениям в базе данных. Позднее вам очень непросто будет вспомнить, что, например, означает число «1», поэтому

определенно стоит немного задержаться и составить подробный список значений. Это особенно верно, если вы собираетесь сортировать экземпляры модели по значению поля, а в нашем примере мы собираемся реализовать такую возможность.

Кроме того, мы сможем использовать эти значения в общедоступных представлениях, чтобы определить, что будет доступно для просмотра нашим посетителям, – то есть они смогут просматривать статьи с признаками «Published» (опубликовано) и «Archived» (перемещено в архив), но не статьи с признаками «Needs Edit» (требует доработки) и «Needs Approval» (требует утверждения). Это обычная практика, определяемая требованиями бизнеса, проекта и/или приложения.

Если вдруг выяснится, что перечень возможных вариантов выбора, подобный этому, не укладывается в жестко определенный список, попробуйте вместо него использовать поле `ManyToManyField`, которое может играть ту же роль, но при этом у вас появится возможность редактировать список вариантов в приложении администрирования, как любые другие данные.

Вслед за определением `STATUS_CHOICES` следуют определения полей.

- `title`: Заголовок, который будет отображаться в заголовке окна броузера и в заголовке отображаемой страницы.
- `slug`: Уникальное имя страницы, которая соответствует этому адресу URL. Это лучше, чем использование простых целочисленных значений первичного ключа.
- `category`: Категория данного элемента содержимого. Это внешний ключ на другую модель, определение которой приводится ниже.
- `markdown_content`: Содержимое страницы в формате Markdown (подробнее об этом формате рассказывается ниже).
- `html_content`: Содержимое страницы в формате HTML. Это содержимое будет автоматически отображаться во время редактирования, поэтому при отображении страниц не будет затрачиваться дополнительное время на ее преобразование. Чтобы уменьшить вероятность путаницы, это поле недоступно для прямого редактирования (и потому не будет отображаться в форме редактирования в приложении администрирования).
- `owner`: Пользователь с правами администратора (или, в терминах платформы Django, внешний ключ на объект `User`), «владеющий» этим элементом содержимого.
- `status`: Признак состояния элемента в терминах производственного процесса.
- `created`: Время создания элемента; автоматически присваивается текущее время (с помощью модуля `datetime` из стандартной библиотеки языка Python).

- modified: Время последнего изменения элемента; первоначально присваивается текущее время. Мы будем предпринимать определенные шаги, чтобы обновлять значение этого поля при редактировании элемента. Значение этого поля будет отображаться на страницах просмотра статей.

В этой модели мы добавили один косметический штрих только для пользователей приложения администрирования – атрибут verbose_name_plural во вложенном классе Meta. Это предупреждает возможность отображения некорректно образованного имени – «Storys». Наконец, у нас имеется метод get_absolute_url, декорированный функцией permalink, упоминавшейся в главе 7 «Фотогалерея».

Импортирование

Помимо импортирования привычного модуля django.db.models (и связывания декоратора permalink, о чем будет говориться ниже) необходимо также импортировать модуль datetime (с помощью которого будут заполняться поля created и modified) и модель User, которая определяется во вспомогательном приложении contrib.auth. Последним импортируется модуль администрирования, используемый для регистрации наших моделей в приложении администрирования.

```
import datetime
from django.db import models
from django.db.models import permalink
from django.contrib.auth.models import User
from django.contrib import admin
```

Перейдя к разработке более совершенных приложений на платформе Django, вы можете обнаружить некоторые недостатки модели User. Например, поддерживаемое в модели представление имени пользователя может не соответствовать вашим требованиям. Тем не менее, модель User представляет собой достаточно удобное, адекватное и законченное решение и чрезвычайно удобна для использования во многих настоящих приложениях.

Заключительная модель

Итак, модель User импортируется прямо из вспомогательного приложения «auth», входящего в состав Django. А модель Category? Это наша собственная модель. Определение модели, которое приводится ниже, следует поместить в файл models.py непосредственно перед определением модели Story.

```
class Category(models.Model):
    """Категория содержимого"""
    label = models.CharField(blank=True, max_length=50)
    slug = models.SlugField()

    class Meta:
```

```
        verbose_name_plural = "categories"

    def __unicode__(self):
        return self.label

class CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug': ('label',)}

admin.site.register(Category, CategoryAdmin)
```

Модель Category проста до тривиальности. В приложениях на платформе Django часто можно встретить такие простые модели, а иногда даже еще проще – содержащие единственное поле. Тот же самый эффект можно было бы получить, добавив поле «category» в модель Story, но это могло бы некоторые вещи усложнить (переименование категорий), а некоторые сделать невозможными (добавление атрибутов к категориям, например описания). Платформа Django настолько упрощает создание реляционных моделей, что практически всегда имеет смысл поступать именно таким образом.

Как и в модели Story, мы определили здесь атрибут verbose_name_plural, чтобы не выглядеть безграмотными перед пользователями приложения администрирования.

Управление доступностью статей для просмотра

В нашей базе данных хранятся как опубликованные (значения 3 и 4 в кортеже STATUS_CHOICES), так и неопубликованные (значения 1 и 2) статьи. Нам необходим некоторый удобный способ обеспечить доступность для просмотра только опубликованных статей в общедоступном разделе сайта и всех статей – в административном разделе. Поскольку это вопрос логики работы приложения, а не стиля представления, это должно быть реализовано в модели.

Мы *могли бы* реализовать это с помощью тегов {% if ... %} в шаблонах, но такое решение в конечном счете привело бы к созданию нестабильных и чересчур раздутых шаблонов. (Если не верите – попробуйте; недостатки такого подхода станут очевидными еще до того, как вы закончите!) Согласно коллективному опыту авторов книги *всегда* стоит отделять логику работы приложения от шаблонов, потому что спустя некоторое время они могут превратиться в спагетти!

Мы добавим эту возможность в нашу модель Story с помощью собственного подкласса класса Manager. Подробнее об этом приеме можно прочитать в разделе «Расширение механизма ORM с помощью собственных подклассов Manager» главы 11 «Передовые приемы программирования в Django». Добавьте следующий программный код в ваш файл models.py, сразу вслед за инструкциями import:

```
VIEWABLE_STATUS = [3, 4]

class ViewableManager(models.Manager):
    def get_query_set(self):
```

```
        default_queryset = super(ViewableManager, self).get_query_set()
        return default_queryset.filter(status__in=VIEWABLE_STATUS)
```

Сначала мы определили переменную VIEWABLE_STATUS как простой список целых чисел, соответствующих значениям состояний статей, доступных для просмотра в общедоступном разделе сайта. Эта переменная является атрибутом модуля, и потому она будет доступна для любых методов, которые мы, возможно, добавим в будущем.

Затем необходимо создать экземпляры объектов-менеджеров внутри модели. В конец вашего файла models.py вслед за определением вложенного класса Meta добавьте следующие две строки, не забудьте при этом оформить отступы, чтобы они принадлежали классу Story.

```
admin_objects = models.Manager()
objects = ViewableManager()
```

Как будет сказано в четвертой части книги, так как менеджер admin_objects определяется первым, он становится менеджером модели по умолчанию и будет использоваться приложением администрирования – этим обеспечивается доступность всех статей для редактирования. Само имя может быть произвольным – оно служит только для напоминания о возлагаемой на этот менеджер функции.

Затем создается экземпляр нашего собственного менеджера, которому присваивается типичное имя objects. Так как это имя используется в файле URLconf и в представлениях, все общедоступные страницы автоматически будут принимать специальные отфильтрованные запросы на извлечение статей, предоставляемые объектом класса ViewableManager.

Работа с форматом Markdown

В завершение мы переопределяем встроенную функцию save, чтобы к тексту, который вводится пользователями с помощью приложения администрирования, применить легковесный язык разметки Markdown. Язык разметки Markdown, отдаленно напоминающий синтаксис Wiki, представляет собой упрощенную альтернативу создания содержимого веб-страниц. Тексты в формате Markdown редактировать удобнее, чем в «сыром» формате HTML; кроме того, этот формат знаком любому, кто составлял электронные письма в простом текстовом формате или редактировал страницы Wiki.

Точно так же можно было бы использовать Textile, ReStructuredText или другие легковесные языки разметки. Суть приема заключается в том, чтобы переопределить метод save модели и реализовать «автоматическое» преобразование разметки Markdown в разметку HTML, благодаря чему ликвидируется необходимость выполнять преобразование при каждом обращении к странице – мы упоминали об этом выше, когда описывали поля markdown_content и html_content.

Почему не WYSIWYG?

Учитывая, что целевой аудиторией системы управления содержимым веб-страниц являются пользователи, как правило, не имеющие специальной технической подготовки, некоторым может показаться немного странным наш выбор языка разметки Markdown. Справедливо. Действительно, существует возможность интегрировать различные HTML-редакторы WYSIWYG (What You See Is What You Get – что видишь, то и получишь) в приложение администрирования платформы Django, что в большей степени соответствовало бы требованиям пользователей.

Один из недостатков такого подхода, помимо дополнительных усилий по реализации, состоит в том, что компонент ввода текста в режиме полного визуального соответствия (WYSIWIG) *все-таки* не в состоянии превратить веб-броузер в Microsoft Word, а кроме того, вы можете столкнуться с проблемой несовместимости броузеров. Однако, с другой стороны, применение таких инструментов может повысить привлекательность систем управления содержимым среди пользователей, не имеющих специальной технической подготовки. На сайте *withdjango.com* вы найдете рекомендуемые дополнения, обеспечивающие возможность редактирования в режиме WYSISYG, и другие советы.

Чтобы обеспечить возможность обработки текста в разметке Markdown на языке Python, вам сначала необходимо загрузить модуль Python-Markdown, так как он не является частью стандартной библиотеки. Найти этот модуль можно по адресу: *http://www.freewisdom.org/projects/python-markdown/*. После его установки импортируйте из модуля `markdown` функцию `markdown`, как показано ниже:

```
from markdown import markdown
```

Инструкция может показаться тавтологичной, но в действительности для языка Python совершенно обычно, когда модуль и атрибут, импортируемый из модуля, носят одно и то же имя.

Незнание языка разметки Markdown ничуть не повлияет на вашу способность понять работу этого приложения; тем не менее, для непосвященных ниже приводится несколько примеров. При желании вы можете опробовать их у себя. В данном демонстрационном примере мы определяем вспомогательную функцию `tidy_markdown`, которая просто удаляет символы перевода строки (`\n`), вставляемые функцией `markdown`. (При использовании модуля `markdown` для воспроизводства более объемных фрагментов разметки HTML эти символы перевода строки препятствуют выводу текста в одну длинную строку.)

```
>>> from markdown import markdown
>>> def tidy_markdown(text):
...     return markdown(text).replace('\n', '')
>>>
>>> tidy_markdown("Hello")
'<p>Hello</p>'
>>> tidy_markdown("# Heading Level One")
'<h1>Heading Level One</h1>'
>>> tidy_markdown("Click here to buy my book (<http://withdjango.com/)">)
'<p><a href="http://withdjango.com/">Click here to buy my book</a></p>'
>>> tidy_markdown("""
... An alternate H1 style
... ====================
... > A blockquote
... * Bulleted item one
... * Bulleted item two
... """)
'<h1>An alternate H1 style</h1><blockquote><p>A blockquote</p></
blockquote><ul>
<li>      Bulleted item one </li>
<li>      Bulleted item two </li></ul>'
```

Как видите, функция получает простой текст в разметке Markdown и возвращает разметку HTML.

Итак, вернемся к нашему приложению на платформе Django: чтобы обеспечить автоматическое преобразование содержимого в разметке Markdown в формат HTML во время сохранения, нам необходимо дополнить реализацию нашей модели. Эту простую функцию из трех строк нужно поместить непосредственно перед инструкцией присваивания `admin_object` (с тем же уровнем отступов, что и остальная часть определения класса модели).

```
def save(self):
    self.html_content = markdown(self.markdown_content)
    self.modified = datetime.datetime.now()
    super(Story, self).save()
```

Когда наш программный код (или любое другое приложение, работающее с нашей моделью, такое как приложение администрирования платформы Django) попытается сохранить объект в базе данных, в первую очередь будет вызван метод `save` модели, который преобразует введенный пользователем текст в формате Markdown в формат HTML. (Если вам необходимо освежить в памяти синтаксис вызова функции `super`, обратитесь к главе 1, «Практическое введение в Python для Django».)

Читателей, обладающих опытом работы с базами данных, может покоробить присутствие поля, содержимое которого легко можно воспроизвести на основании содержимого другого поля. Если бы для выполнения преобразований не требовалось тратить время на вычисления,

нам не потребовалось бы сохранять отображаемую разметку HTML. Это обычная плата за скорость, которая различна для разных проектов. Мы предполагаем, что здесь вычислительные ресурсы являются ограничивающим фактором, как и в случае любых сайтов с большим объемом трафика и необязательно большим объемом содержимого. Для сайтов, где требуется минимизировать объем базы данных, таких как сайты, содержащие форумы сообществ с тысячами и миллионами записей, создание представления страницы при каждом просмотре может оказаться более правильным выбором.

Так как поле, хранящее разметку HTML, помечено как `editable=False`, оно не отображается в интерфейсе административного раздела сайта. Это предотвращает возможность правки отображаемой разметки HTML, которая будет перезаписана щелчком на кнопке Save (сохранить), и обеспечивает более очевидный способ взаимодействия с пользователем. Все изменения, произведенные в исходной разметке Markdown, будут преобразованы в разметку HTML и сохранены в поле `html_content` без дополнительного вмешательства. Во время сохранения также выполняется запись значения текущего времени в поле `modified`.

Дополнительную информацию о языке разметки Markdown и его синтаксисе вы найдете на официальном сайте *http://daringfireball.net/projects/markdown/*. Кроме того, следует знать, что модуль Python-Markdown распространяется вместе с несколькими интересными расширениями. Сама данная книга была *написана* на языке разметки Markdown с применением расширения Wrapped Tables – `wtables` (за дополнительной информацией обращайтесь по адресу *http://brian-jaress.livejournal.com/5978.html*)! Существует еще один проект на языке Python для работы с разметкой Markdown, с которым вам следует познакомиться, – ищите его по адресу *http://code.google.com/p/python-markdown2/*.

Шаблоны адресов URL в urls.py

Переопределением метода `save` мы закончили описание наших моделей. Но прежде чем мы перейдем к функциям представлений и шаблонам, необходимо настроить адреса URL. Ниже приводится содержимое файла `urls.py` уровня проекта.

```
urlpatterns = patterns('',
    url(r'^admin/(.*)', admin.site.root),
    url(r'^cms/', include('cmsproject.cms.urls')),
)
```

Строка с определением шаблона адреса приложения администрирования та же, что и обычно, а второй шаблон определяет префикс «cms/» всех адресов URL внутри приложения системы управления содержимым. Если вам потребуется определить другой префикс – например, «stories» или «pages», вы можете определить его здесь же. В главе 7

описывается альтернативный прием реализации гибкой схемы адресов URL.

Ниже приводится содержимое файла urls.py уровня приложения, присутствующего в вызове функции include выше:

```
from django.conf.urls.defaults import *
from cms.models import Story

info_dict = {'queryset': Story.objects.all(), 'template_object_name': 'story'}

urlpatterns += patterns('cmsproject.cms.views',
    url(r'^category/(?P<slug>[-\w]+)/$', 'category', name="cms-category"),
    url(r'^search/$', 'search', name="cms-search"),
)

urlpatterns = patterns('django.views.generic.list_detail',
    url(r'^(?P<slug>[-\w]+)/$', 'object_detail', info_dict, name="cms-story"),
    url(r'^$', 'object_list', info_dict, name="cms-home"),
)
```

По порядку: наши определения обеспечивают отображение страниц с отдельными статьями, с полными списками статей, со списками статей по категориям и со списками, содержащими результаты поиска.

Поскольку в этом приложении мы также используем универсальные представления Django, эти определения являются тем самым местом, где выполняется основная работа приложения. Мы определили четыре шаблона адресов URL, разделив их на два объекта patterns в соответствии с различиями в префиксах; при этом мы могли бы также импортировать функции представлений напрямую и использовать их непосредственно.

Примечание

Мы решили использовать здесь строки с именами функций представлений, потому что нам необходимо будет иметь возможность задействовать приложение администрирования и универсальные представления еще до того, как мы определим свои специализированные представления – попытка импортировать несуществующие функции в этой ситуации не привела бы к успеху. Использование строк и/или объектов функций в определениях адресов URL – это во многом вопрос личных предпочтений; вы можете использовать тот из подходов, который лучше подходит в вашем случае.

Как было показано во многих примерах из предыдущих глав, универсальные представления имеют массу необязательных аргументов, управляющих их поведением. Здесь мы использовали единственный такой аргумент – template_object_name, который позволяет ссылаться в шаблоне на объект статьи по имени story, а не по имени object, используемому по умолчанию.

Представления административного раздела

Теперь вы должны были получить работающий административный раздел сайта с приложением системы управления содержимым. (Не забудьте предварительно запустить команду `manage.py syncdb`, чтобы создать таблицы в базе данных.) Подключитесь к нему. На рис. 8.4 показано, как должна выглядеть страница администратора после того, как вы зарегистрируетесь. На рис. 8.5 показано, как выглядит страница Add Story (Добавить статью) после щелчка на кнопке Add (Добавить).

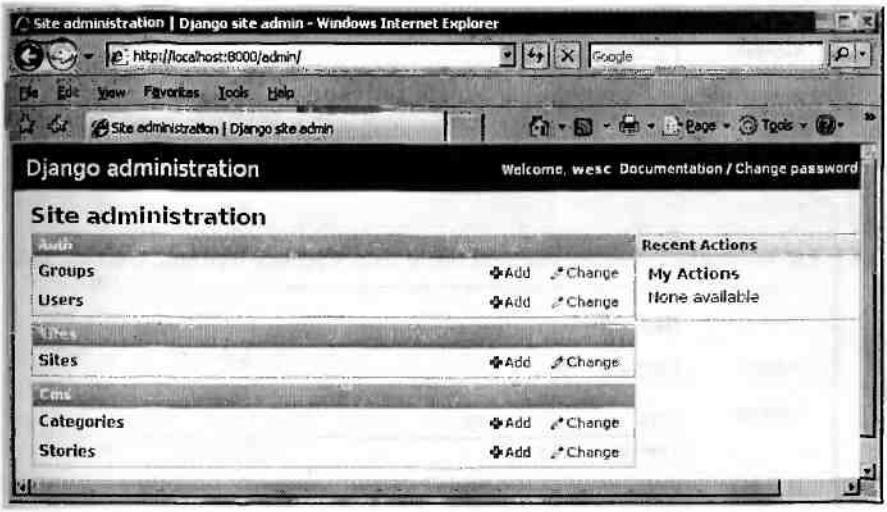

Рис. 8.4. Страница администратора

Теперь вы можете создать категорию. Если на странице Add Story (Добавить статью) щелкнуть на ярлыке «+», появится окно диалога, как показано на рис. 8.6.

Например, введите название «Site News» (новости сайта) в поле Label (Метка). Вы увидите, как одновременно с этим в поле Slug (Ключ) появится строка, определяющая путь к разделу сайта (рис. 8.7).

Теперь можно продолжить и закончить добавление статьи В нашем примере мы установили статус статьи как «Published» (опубликовано) (рис. 8.8).

Сразу после сохранения будет выполнено перенаправление на страницу Story (статья) (рис. 8.9), где будет должна появиться информация о статье.

Теперь можно добавить и/или отредактировать еще несколько статей, хотя бы по одной для состояний «Published» (опубликовано) и «Archived» (перемещено в архив), чтобы было на что посмотреть на основном сайте!

9 Зак. 1496

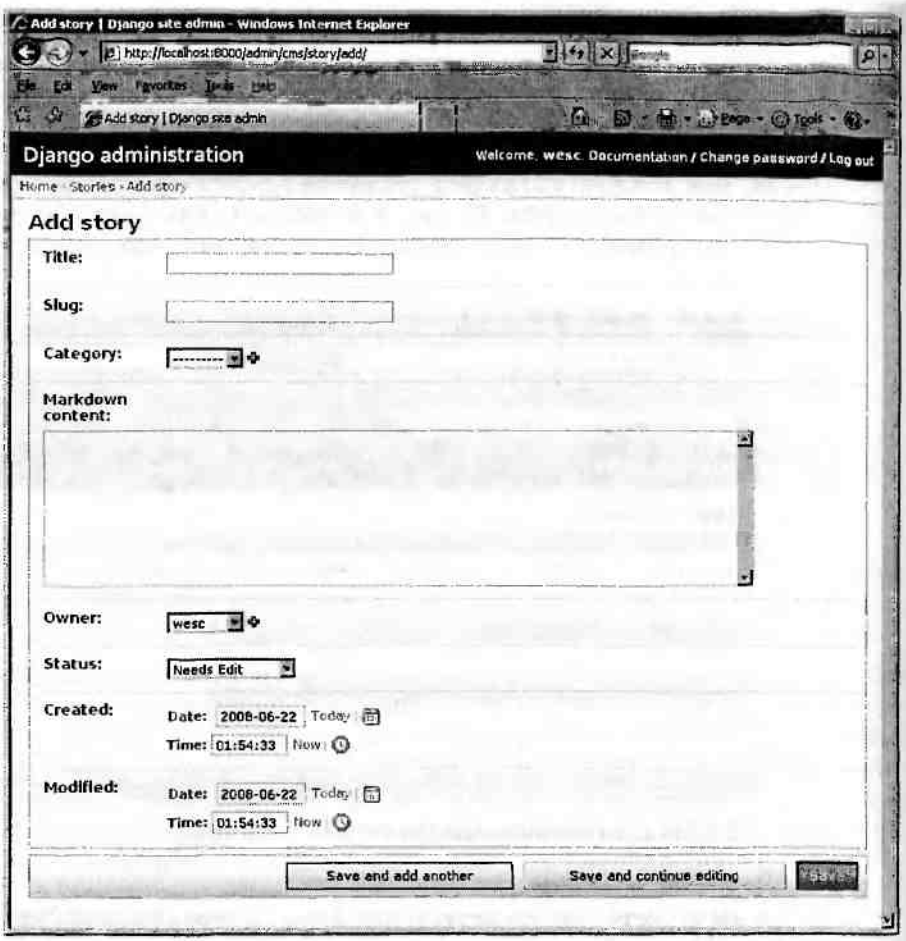

Рис. 8.5. Добавление статьи на странице администратора

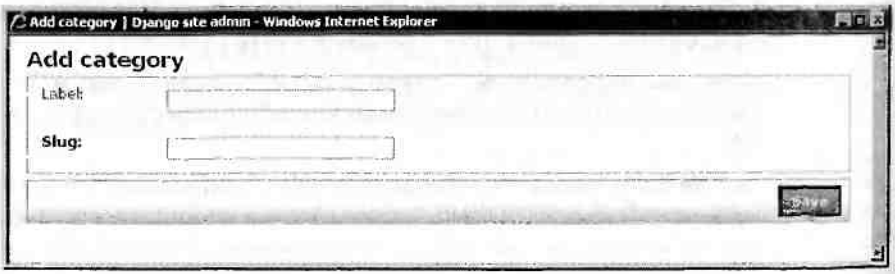

Рис. 8.6. Добавление категории в процессе добавления статьи

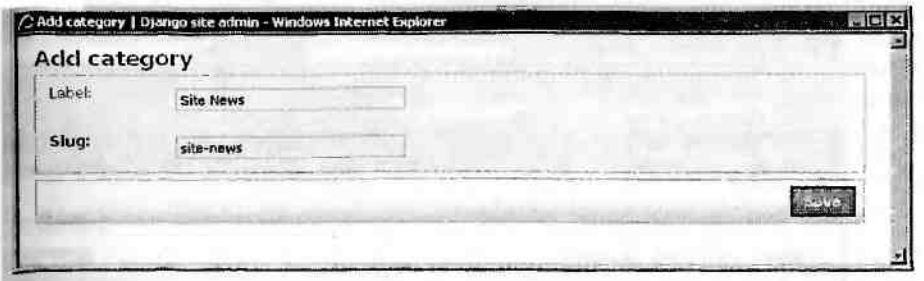

Рис. 8.7. Добавление категории «Site News»

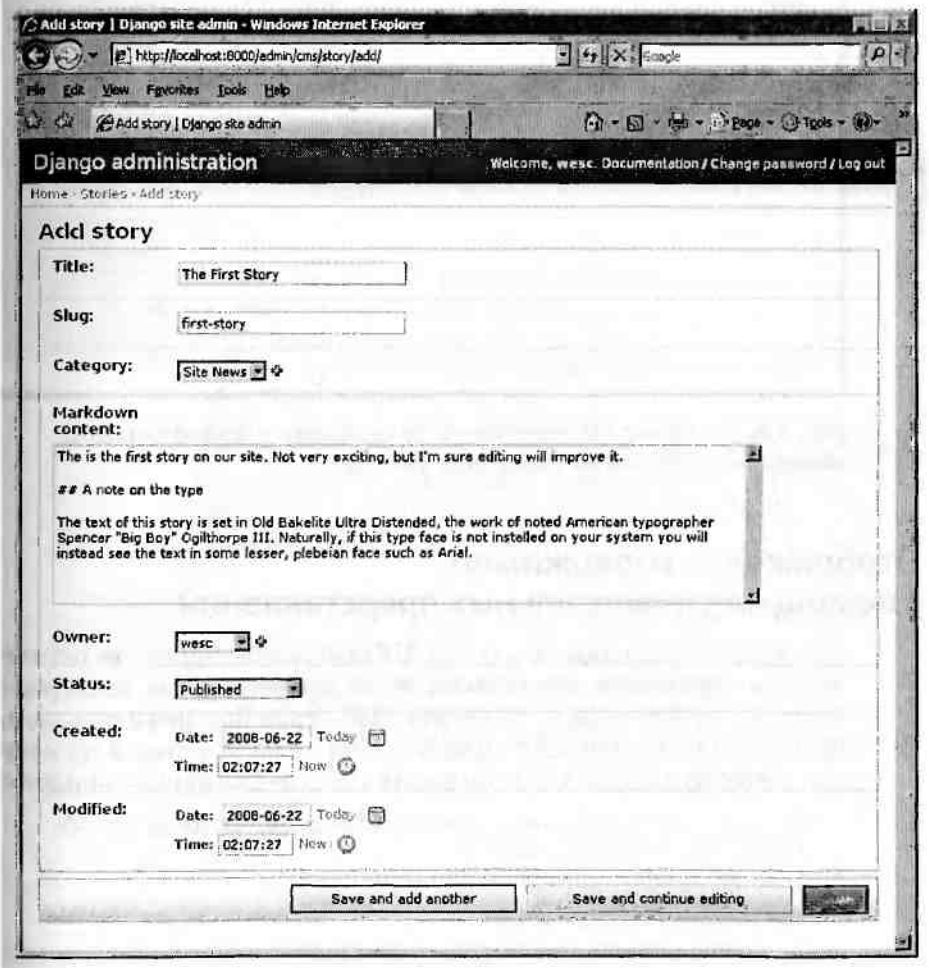

Рис. 8.8. Завершение операции добавления первой статьи

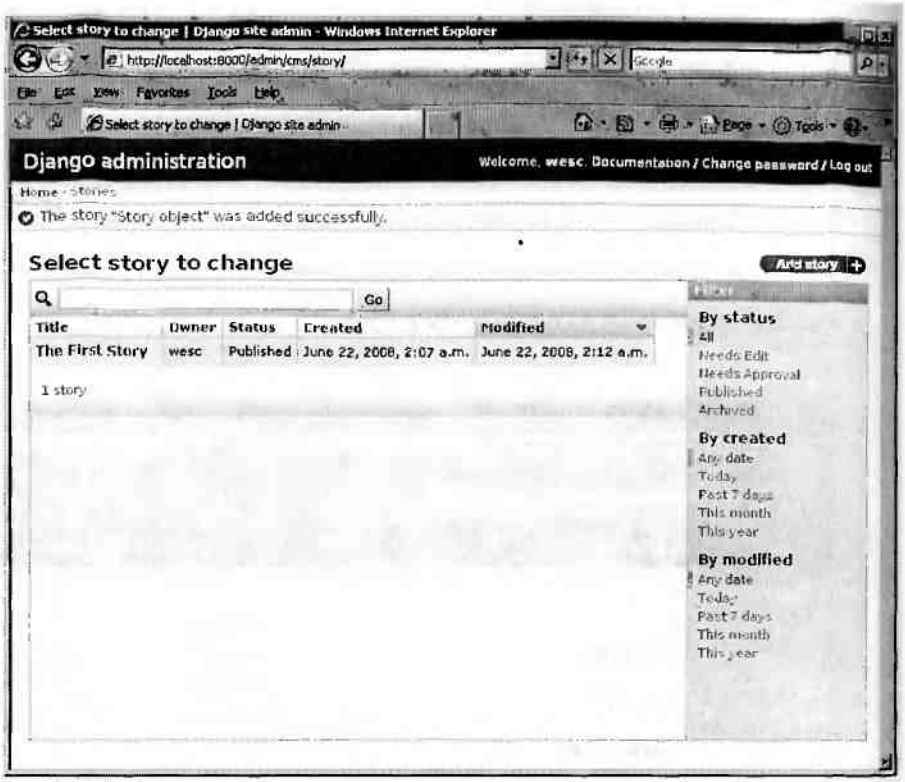

Рис. 8.9. Просмотр списка статей на странице администратора; обратите внимание на доступные фильтры

Отображение содержимого с помощью универсальных представлений

Как можно было видеть в файле URLconf, в большинстве случаев для нужд отображения мы использовали универсальные представления. Однако нам необходимо написать пару строк программного кода, чтобы реализовать функцию представления списков статей по категориям. Ниже приводится начало файла views.py для нашего приложения.

```
from django.shortcuts import render_to_response, get_object_or_404
from django.db.models import Q
from cms.models import Story, Category

def category(request, slug):
    """По заданному ключу категории отображает все элементы этой категории"""
    category = get_object_or_404(Category, slug=slug)
    story_list = Story.objects.filter(category=category)
    heading = "Category: %s" % category.label
    return render_to_response("cms/story_list.html", locals())
```

Как видите – это обычная функция представления, но она реализует действия, которые не могут быть выполнены ни одной из имеющихся функций универсальных представлений, и именно по этой причине нам потребовалось написать свою функцию. Далее мы переходим к нашим шаблонам, а потом рассмотрим второе наше представление, реализующее интерфейс функции поиска.

Шаблоны

Как и в большинстве проектов на платформе Django, у нас имеется базовый шаблон base.html, который наследуется другими шаблонами. В данном случае у нас имеется еще два шаблона: story_detail.html и story_list.html. Создайте все три файла в каталоге cms и добавьте в параметр TEMPLATE_DIRS, который находится в файле settings.py, полный путь к вашему проекту.

Начнем с простого базового шаблона, который выглядит, как показано ниже:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
    <head>
        <title>{% block title %}{% endblock %}</title>
        <style type="text/css" media="screen">
            body { margin: 15px; font-family: Arial; }
            h1, h2 { background: #aaa; padding: 1% 2%; margin: 0; }
            a { text-decoration: none; color: #444; }
            .small { font-size: 75%; color: #777; }
            #header { font-weight: bold; background: #ccc; padding: 1% 2%; }
            #story-body { background: #ccc; padding: 2%; }
            #story-list { background: #ccc; padding: 1% 1% 1% 4%; }
            #story-list li { margin: .5em 0; }
        </style>
    </head>
    <body>
        <div id="header">
            <form action="{% url cms-search %}" method="get">
                <a href="{% url cms-home %}">Home</a> &bull;
                <label for="q">Search:</label> <input type="text" name="q">
            </form>
        </div>
        {% block content %}
        {% endblock %}
    </body>
</html>
```

Отдельные элементы этого шаблона мы опишем ниже в этой главе. А теперь взгляните на шаблон отображения отдельной статьи (story_detail.html), который, как уже говорилось выше, наследует базовый шаблон.

```
{% extends "cms/base.html" %}
{% block title %}{{ story.title }}{% endblock %}
{% block content %}
    <h1>{{ story.title }}</h1>
    <h2><a href="{% url cms-category story.category.slug %}">{{
story.category
}}</a></h2>
    <div id="story-body">
        {{ story.html_content|safe }}
        <p class="small">Updated {{ story.modified }}</p>
    </div>
{% endblock %}
```

Это, пожалуй, самый простой из полезных шаблонов – он принимает единственную переменную шаблона – story. Шаблон будет успешно работать с любым объектом, имеющим атрибуты title и html_content.

Одним из важнейших элементов шаблона является фильтр safe, который применяется к содержимому поля html_content. По умолчанию платформа Django автоматически экранирует в шаблонах все служебные символы HTML, чтобы обезопасить себя от злонамеренного содержимого, введенного пользователем (рост нападений на веб-приложения вызывает серьезную обеспокоенность). Так как наш исходный текст в формате Markdown вводится доверенными пользователями, мы можем пометить содержимое как «безопасное» (safe) и позволить броузеру буквально интерпретировать разметку HTML, вместо того чтобы экранировать, например , в <em> и т. д.

Наш шаблон списка статей story_list.html используется несколькими различными представлениями, которым необходимо отобразить список из нескольких статей – списки по категориям, результаты поиска и домашняя страница.

```
{% extends "cms/base.html" %}
{% block content %}
    {% if heading %}
        <h1>{{ heading }}</h1>
    {% endif %}
    <ul id="story-list">
    {% for story in story_list %}
        <li><a href="{{ story.get_absolute_url }}">{{ story.title }}</a></li>
    {% endfor %}
    </ul>
{% endblock %}
```

Он немногим сложнее шаблона отображения одной статьи. Он в цикле выполняет обход элементов в объекте story_list, для каждого элемента списка создает тег <li>, в который помещает текст заголовка статьи в виде ссылки на страницу со статьей.

Отображение статей

Благодаря использованию ключей статей адрес URL первой статьи на нашем сервере разработки будет иметь вид: *http://localhost:8000/cms/first-story/*. Проверьте, был ли перезапущен сервер, введите URL в адресную строку броузера, и в результате вы должны увидеть страницу, напоминающую ту, что изображена на рис. 8.10.

Теперь проверим работу представления `object_list`, отображающего домашнюю страницу сайта. Адрес этой страницы: *http://localhost:8000/cms/*. Когда вы перейдете по этому адресу, вы должны увидеть нечто похожее на рис. 8.11.

Заголовки статей в этом списке являются ссылками, сгенерированными методом `get_absolute_url`, который мы создали ранее.

Обратите внимание: на странице имеется поле ввода, реализующего функцию поиска! Читайте дальше, чтобы стало понятно, как мы заставили его работать.

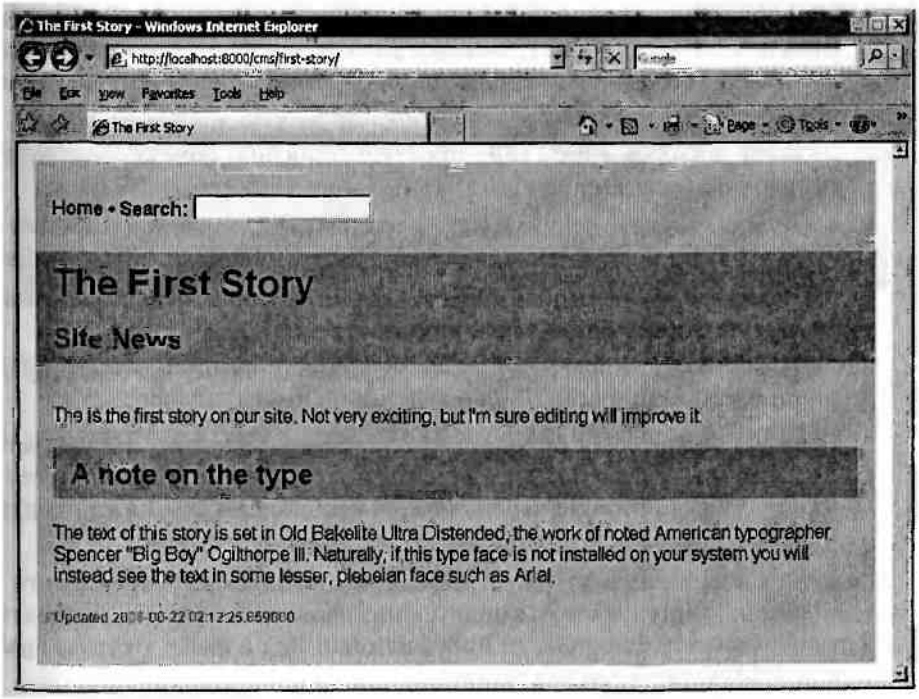

Рис. 8.10. Страница с нашей первой статьей

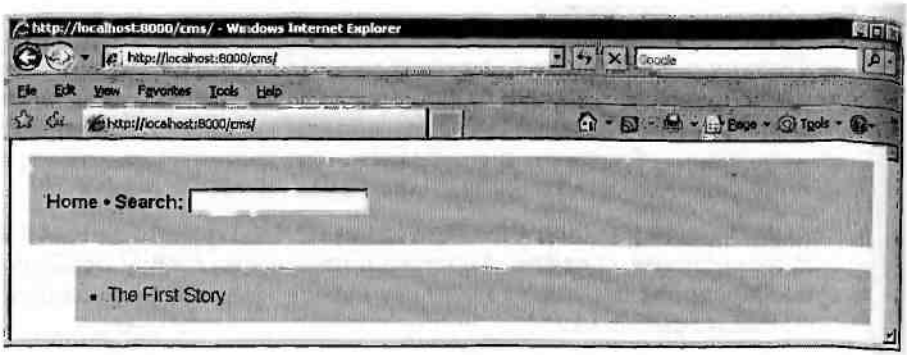

Рис. 8.11. Домашняя страница сайта с полным списком статей

Добавление функции поиска

Возможность поиска в текстовом содержимом является насущной необходимостью. Для общедоступного сайта всегда можно добавить окно Google SiteSearch (*http://www.google.com/coop/cse/*), но гораздо приятнее иметь больший контроль над процессом поиска и представлением результатов.

Давайте добавим в наш сайт простую функцию поиска. Для этого нам потребуется совсем немного:

- Добавить в шаблон `base.html` форму HTML, содержащую поле ввода искомой строки, чтобы оно отображалось на каждой странице
- Реализовать функцию представления, которая будет принимать искомую строку из формы и искать статьи
- Шаблон `story_list.html`, уже созданный нами, который будет использоваться для отображения результатов

Если вы вернетесь к определению шаблона `base.html`, вы увидите, что он включает в себя поле поиска, расположенное в верхней части страницы. Чтобы это поле *действовало*, необходима функция представления, которая обрабатывала бы форму после ее отправки.

Эта задача не может быть решена с помощью универсального представления, поэтому нам придется написать свою функцию представления. Добавьте следующий программный код в файл `views.py` сразу после определения метода `category`:

```
def search(request):
    """
    Возвращает список статей, содержащих искомую строку
    в заголовке или в основном тексте.
    """
    if 'q' in request.GET:
        term = request.GET['q']
```

```
        story_list = Story.objects.filter(Q(title__contains=term) |
                                          Q(markdown_content__contains=term))
        heading = "Search results"
    return render_to_response("cms/story_list.html", locals())
```

Это специализированное представление, но оно не требует наличия собственного шаблона. Мы можем повторно использовать наш шаблон `story_list.html` – при условии, что ему будет передаваться то, что он ожидает, – объект `QuerySet`, содержащий объекты модели `Store` в контекстной переменной с именем `story_list`. Алгоритм поиска чрезвычайно прост – объект `Story` считается соответствующим, если текст, переданный через форму поиска, полностью присутствует в заголовке или в разметке Markdown.

Давайте добавим еще несколько «статей». В нашем примере мы добавили страницу «About Us» (о нас) (такую же, как в примере с использованием приложения Flatpages) и пометили ее как «Archived». Затем мы добавили страницу «Contact Us» (как с нами связаться), но оставили ее в состоянии «Needs Edit» (требует доработки). Наша административная страница теперь отображает все три статьи и их статус, как показано на рис. 8.12.

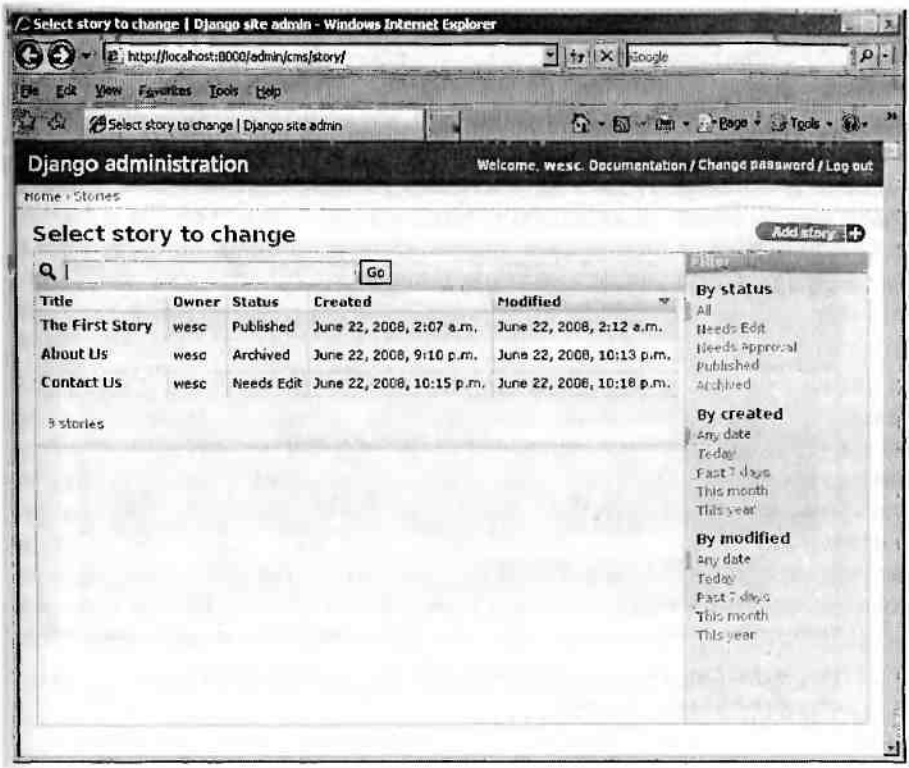

Рис. 8.12. Административная страница с полным списком статей

Нам требуется, чтобы на главной странице общедоступного раздела отображались только ссылки на страницы со статьями, разрешенными для общего доступа (с признаками «Published» и «Archived», как определено в списке VIEWABLE_STATUS). Открыв главную страницу, вы можете убедиться, что страница «Contact Us» отсутствует в списке, как показано на рис. 8.13!

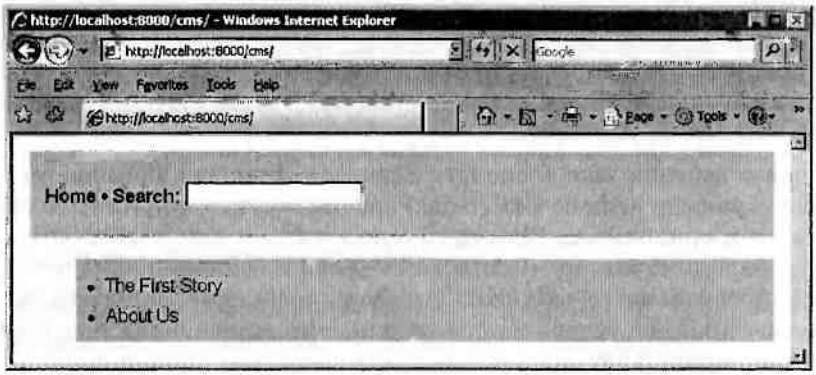

Рис. 8.13. Главная страница со списком статей, доступных для общего просмотра

Давайте теперь опробуем нашу функцию поиска. Выполнив поиск слова «typographer» (печатник), мы обнаружили только один совпадающий документ – с нашей первой статьей, как показано на рис. 8.14.

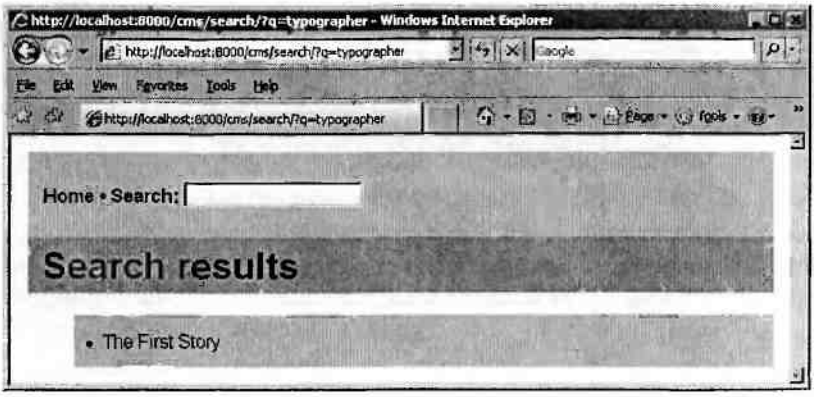

Рис. 8.14. Страница с результатами поиска (кроме того, это список общедоступных статей)

На этом мы завершаем обзор базовой функциональности, реализованной в нашей системе управления содержимым. Давайте теперь обсудим заключительные аспекты поведения нашего приложения: управление пользователями, систему прав доступа и обеспечение производственного процесса, которые диктуются логикой управления.

Управление пользователями

Наша система реализует понятие владения – каждая статья ассоциирована с определенным объектом `User`. Нет никаких технических препятствий к изменению или удалению содержимого пользователями, которые не являются владельцами этого содержимого, точно так же фактически нет никаких технических препятствий к изменению поля, определяющего владельца. Наличие этого поля не создает новых прав управления объектами – которых не существовало прежде.

Примечание

В ближайшем будущем в Django появится возможность реализовать более точное распределение прав доступа к объектам с помощью новых средств администрирования, которые на момент написания этих строк еще находились на стадии разработки. За дополнительной информацией по этой теме обращайтесь на сайт *withdjango.com*.

Тем не менее, такой вид неофициального или добровольного владения по-прежнему может быть полезным в организациях, где существует атмосфера взаимного доверия. В действительности это мало чем отличается от обычной работы в офисе, где вы уверены, что сослуживцы не утащат ваш особый красный степлер и не порвут документ, который лежит у вас на столе. Одно из удобств реализации такой схемы владения состоит в том, что мы использовали встроенную в Django модель `User`. Нам не пришлось добавлять никакого кода, описывающего модель. Таким образом, мы управляем пользователями с помощью приложения администрирования, входящего в состав Django.

Будучи суперпользователем в приложении администрирования, вы можете с его помощью определять, кто будет обладать правом управления пользователями и группами и кто из них будет иметь доступ к модели `Story`. Точно так же можно дать пользователям возможность изменять объекты `Story`, но не `Category`. Такое ограничение может оказаться вполне разумным, так как большинство редакторов содержимого должны не производить реорганизацию структуры информационного наполнения сайта, а только добавлять новые и обновлять существующие элементы содержимого.

Поддержка производственного процесса

Ниже приводится простой порядок следования стадий производственного процесса, на основе которого выбирались значения для поля состояния.

1. Внештатный автор или сотрудник присылает содержимое для страницы. Это содержимое находится в состоянии чернового варианта и требует доработки.
2. После того как документ пройдет этап редактирования, он должен быть утвержден к публикации.
3. Как только статья будет помечена как «опубликованная», она станет доступна для общего просмотра.
4. Если статья устаревает, она помечается как «перемещенная в архив». Это может означать, что статью можно будет отыскать с помощью функции поиска по сайту, но она не будет отображаться в разделе «Последние статьи» на главной странице.

Пример этой главы не предусматривает расширения возможностей приложения администрирования. Но при желании можно было бы создать специализированные представления, использующие поле состояния, – например, выделение элементов цветом в зависимости от стадии, на которой они находятся, или предоставление пользователям списков доступных операций после регистрации в административном разделе.

Примечание

Подробнее о расширении возможностей приложения администрирования рассказывается в главе 11 «Передовые приемы программирования в Django».

Обратите внимание на использование в нашем файле `models.py` особенности `list_filter` приложения администрирования, которая обеспечивает удобный способ выбора статей по любой из четырех категорий. Например, редактор может использовать эти фильтры для выбора всех статей с признаком «Need Edit» (требует доработки) или стажер, которому поручено отсортировать устаревшие материалы, может использовать фильтры для выбора статей с признаком «Archived» (перемещено в архив).

Возможные улучшения

Как уже говорилось в начале главы, число архитектур систем управления содержимым совпадает с числом пользователей. Пример приложения, которое вы создали в этой главе, можно было бы расширять в разных направлениях в зависимости от того, какие возможности необходимы. Ниже приводится несколько идей.

Постраничный просмотр. Страницы со списками статей более или менее просты в управлении, пока число статей не превышает нескольких десятков. Но, как только вы перевалите рубеж нескольких сотен элементов, попытка отобразить их все на одной странице может привести пользователя в замешательство, а также повлиять на производительность сайта в целом. К счастью, платформа Django предлагает некоторую поддержку постраничного вывода, главным образом средствами модуля `django.core.paginator`. За дополнительной информацией об этом модуле обращайтесь к официальной документации.

Более мощная функция поиска. Наша функция поиска вполне удобна, но она не предлагает той широты возможностей, которые предоставляют поисковые механизмы во Всемирной паутине – например, фразы, состоящие из нескольких слов, в идеале должны рассматриваться как коллекции независимых слов, если не указано обратное. Реализация такого механизма поиска может оказаться очень сложной, поэтому, если вы предполагаете организовать полнотекстовый поиск по большому числу записей, вам наверняка пригодится такой поисковый механизм, как Sphinx, способный интегрироваться в платформу Django. За дополнительной информацией обращайтесь на сайт *withdjango.com*.

Уведомления об изменении состояния. У нас уже имеется специализированный метод `save`, выполняющий преобразование разметки Markdown в разметку HTML. Мы легко могли бы дополнить его возможностью определять момент изменения признака производственной стадии и отправлять по электронной почте уведомления человеку, ответственному за данную стадию. Для этого можно было бы заменить поле состояния на поле внешнего ключа, ссылающегося на полноценную модель `Status`, в которой помимо полей с числовыми и текстовыми значениями из списка `STATUS_CHOICES` присутствовало бы поле `status_owner` – внешний ключ, ссылающийся на модель `User`. Наш метод `save` мог бы сравнивать значение поля `status` в записи с сохраняемым значением и в случае их отличия вызывать функцию `send_mail`, чтобы известить ответственного пользователя.

Динамическая генерация цепочек навигации. Наше приложение не предусматривает решение проблемы навигации по сайту, кроме предоставления по умолчанию списка всех статей. Для настоящего сайта требуется что-то получше. Как вариант, в модель `Story` можно было бы добавить поле, связанное с навигацией. Но более гибкое решение заключается в создании отдельной модели `Navigation`, которая могла бы состоять всего из трех полей: позиция в общей последовательности элементов навигации, метка для отображения пользователю и внешний ключ на статью, на которую должен отсылать данный элемент навигации.

Комментарии пользователей. Наша система управления содержимым прекрасно справляется с публикацией содержимого, но не дает конеч-

ным пользователям возможность оставлять свои комментарии. Было бы вполне естественно предоставить возможность оставлять комментарии к отдельным статьям. К счастью, в платформе Django имеется прекрасная встроенная система комментариев, которая может работать как с зарегистрированными, так и с анонимными пользователями. К сожалению, когда мы работали над этой книгой, система комментариев находилась на стадии полной переработки, поэтому мы не смогли описать ее в этом издании книги. Однако, если ее функциональные возможности представляют для вас интерес, обращайтесь к официальной документации, которая будет обновлена, как только эта система станет доступна.

Статические файлы. Многие торговые организации и организации по связям с общественностью желают иметь возможность выгружать некоторую информацию на сайт, чтобы рассылать ее своим клиентам – как существующим, так и потенциальным, а также обеспечивать доступ к презентациям, отчетам, техническим описаниям и тому подобному – в виде файлов в формате PDF, документов Word, книг Excel, сжатых архивов и т. д.

В заключение

Это была длинная глава, но вы должны были увидеть полномасштабный пример, демонстрирующий, как использовать базовые компоненты платформы Django и вспомогательные приложения, – как для создания сайтов на основе простых «плоских» страниц, так и для построения более сложной системы управления содержимым.

Хотелось бы надеяться, что к настоящему моменту вы гораздо полнее познакомились со способом создания приложений на платформе Django, включающем в себя следующие этапы: создание проекта и приложения с помощью инструментов командной строки, разработку модели (включая использование возможностей приложения администрирования), определение адресов URL, использование универсальных и специализированных представлений и создание иерархии шаблонов.

В этой части книги мы рассмотрим еще два примера приложений: в одном из них для создания живого блога используется технология Ajax, а другой пример представляет собой приложение pastebin на платформе Django.

9

Живой блог

Эта книга посвящена созданию приложений на платформе Django, и платформа, как вы уже видели, обладает достаточно широкими встроенными функциональными возможностями, так что вы можете добиться многого, не выходя за ее пределы. Однако, как и у любого другого инструмента, у платформы Django имеются свои ограничения. Одним из таких явных ограничений является отсутствие поддержки популярной веб-технологии Ajax (Asynchronous JavaScript And XML – асинхронный JavaScript и XML).

К счастью, это в действительности означает, что Django не привязывает вас к какой-то *единственной* библиотеке поддержки Ajax, а просто оставляет дверь открытой, предоставляя свободу выбора библиотеки.

В этой главе мы продемонстрируем относительно простой пример использования технологии Ajax для создания так называемого живого блога. **Живой блог** – это веб-страница со списком коротких записей, способная обновлять свое содержимое без вмешательства пользователя. Те из вас, кто в последние годы следил за событиями, происходящими в мире Apple, могли видеть такие приложения на различных новостных сайтах пользователей Mac, таких как *macrumorslive.com*. Та же концепция в усеченном виде используется в обычных статических блогах, в которых происходящие события представлены в том же формате, но, как правило, не поддерживают функции динамического обновления страницы.

На примере нашего приложения мы рассмотрим все, что необходимо знать для интеграции технологии Ajax в веб-приложения на платформе Django, не углубляясь в специфические особенности организации взаимодействий типа клиент-сервер или воспроизведения анимационных эффектов. Кроме того, мы покажем, насколько хорошо платформа

Django уживается с технологией Ajax, оставаясь при этом независимой от используемых инструментов.

Примечание

Как и в некоторых других примерах приложений, мы будем использовать здесь веб-сервер Apache, чтобы упростить обслуживание статических файлов (в данном случае – сценариев JavaScript).

Что такое Ajax?

Когда кто-то упоминает термин «Ajax», имея в виду веб-технологию, а не чистящее средство, при этом обычно подразумевается два различных, но тесно связанных между собой типа поведения.

- Веб-страницы получают дополнительную информацию, не требуя от пользователя перезагружать страницу или выполнять переход, – подумайте над тем, как сайт GMail отображает различные сообщения, ящики входящих писем и формы без того, чтобы ваш броузер перезагружал и/или полностью перерисовывал страницу.
- Улучшенное «динамическое» поведение пользовательского интерфейса – подумайте над тем, как сайт Google Maps выполняет прокрутку и масштабирование карты или как реализован интерфейс буксировки элементов мышью (drag-and-drop) на различных персональных сайтах, построенных на базе использования «виджетов».

В терминах реализации получение «дополнительной информации» можно представить себе как серию мини-запросов, когда броузер и сервер осуществляют обычный обмен по протоколу HTTP в фоновом режиме, устраняя тем самым необходимость полной перезагрузки страницы. Подробнее о том, как этого добиться, мы поговорим ниже в этой главе, а пока просто отметим, что ответы от сервера приходят обычно в формате XHTML или XML (вот откуда «X» в аббревиатуре AJAX) или в более легковесном формате данных, известном как JSON.

С точки зрения пользовательского интерфейса Ajax – это лишь программный код JavaScript, выполняющий манипуляции с деревом DOM, которые стали возможны благодаря появлению мощных броузеров и клиентских компьютеров. Если учесть возможности отображения корректно оформленных, стилизованных веб-страниц и тот факт, что JavaScript превратился в полноценный язык программирования, можно сказать, что теперь веб-страница напоминает канву для традиционных приемов создания анимационных эффектов в графическом интерфейсе пользователя.

В чем состоит польза Ajax

С точки зрения разработчика, наличие возможности организовать внутри веб-страницы обмен мини-запросами полезно по двум причи-

нам. Во-первых, это помогает экономить полосу пропускания в случаях объемного трафика, потому что клиентские броузеры запрашивают не страницы целиком, а только отдельные порции данных, и, во-вторых, это позволяет создать у пользователя ощущение отзывчивого интерфейса, потому что окно броузера не перерисовывает свое содержимое все время. Это помогает веб-приложениям выглядеть похожими на обычные настольные приложения.

Хотя их иногда и считают всего лишь внешними украшениями, тем не менее, улучшенные анимационные эффекты, механизм буксировки объектов мышью и другие особенности «Web 2.0» могут существенно улучшить восприятие пользователя – при условии, что все эти эффекты не выглядят навязчивыми и используются в умеренных количествах. Вместе с уменьшением числа перезагрузок страницы – благодаря мини-запросам – удачно подобранные анимационные и другие специальные эффекты еще больше стирают грань между веб- и традиционными приложениями с графическим интерфейсом.

Проектирование приложения

Прежде чем перейти к программному коду, выработаем простую спецификацию особенностей, которыми должно обладать наше приложение, и решим, какие инструменты (в частности – какая библиотека Ajax) мы будем использовать для его создания. Для начала выясним некоторые требования и точно определим, что должно делать приложение.

- Приложение должно состоять из единственной веб-страницы. Не должно быть никаких украшений – мы просто создаем сайт, способный обрабатывать единственное событие за один раз.
- Приложение должно отслеживать единственный и непрерывный поток информации. Опять же просто для того, чтобы соблюсти простоту примера.
- «Поток» состоит из параграфов текста, отмеченных датой и временем. То есть нам потребуется создать модель всего с двумя полями.
- Этот поток должен отображаться в порядке, обратном хронологическому, когда самые последние сообщения помещаются в начало списка. Благодаря этому самая свежая информация всегда будет находиться в начале страницы.
- При начальной загрузке страница должна отображать текущее состояние потока. Пользователи, посещающие сайт, должны получить все записи без использования технологии Ajax.
- Страница должна запрашивать новые записи один раз в минуту. Здесь в игру вступает Ajax.
- Новые записи должны создаваться с помощью приложения администрирования. Хотя, с другой стороны, для этих целей легко можно

было бы создать специальную форму – вы могли бы даже организовать отправку сообщений с использованием технологии Ajax, чтобы создать ощущение отзывчивости интерфейса.

Выбор библиотеки Ajax

К моменту написания этих строк было доступно большое число библиотек JavaScript, обеспечивающих поддержку технологии Ajax. Каждая со своими достоинствами, недостатками и преследуемыми целями. Одни из них содержат в себе значительное число виджетов графического интерфейса, другие стремятся максимально упростить создание сценариев на языке JavaScript. Кроме того, они отличаются своими подходами к дополнению JavaScript собственными синтаксическими конструкциями, обеспечивающими манипулирование HTML-структурой веб-страниц.

Большая часть этих библиотек состоит из множества компонентов, доступных для загрузки, – например, «базовые» компоненты, дополняющие JavaScript новыми синтаксическими конструкциями; «сетевые» компоненты; предназначенные для выполнения мини-запросов; «виджеты», используемые для создания элементов графического интерфейса; и, конечно же, имеются «полные» версии, включающие в себя полный пакет элементов. Поэтому, при выборе инструментария вам также потребуется выяснить, для каких целей вы собираетесь его использовать, и загрузить нужную версию.

Это довольно сложная задача, но ее необходимо решить. При использовании крупной и монолитной библиотеки ее необходимо будет загружать при просмотре каждой страницы, что может потребовать немалых ресурсов от веб-сервера. Многокомпонентные же библиотеки при значительном богатстве выбора дают разработчикам возможность включать в свои приложения только необходимые им компоненты.

Ниже приводится краткий перечень наиболее известных библиотек поддержки технологии Ajax и откуда их можно загрузить.

- Dojo: (*dojotoolkit.org*) Библиотека dojo является одной из крупнейших библиотек поддержки Ajax, она состоит из комплекта небольших библиотек и предлагает несколько вариантов загрузки.
- jQuery: (*jquery.com*) Это новейшая библиотека, обеспечивающая мощный синтаксис составления «цепочек» для отбора и воздействия сразу на несколько элементов страницы.
- MochiKit: (*mochikit.com*) Одна из наиболее «Питонических» библиотек JavaScript, создатели которой черпали вдохновение из исходных текстов программ на языке Python и Objective-C.
- MooTools: (*mootools.net*) Библиотека MooTools имеет модульную систему загрузки, позволяющую создавать узкоспециализированные комплекты компонентов библиотеки.

- Prototype: (*prototypejs.org*) Первоначально была разработана в составе платформы разработки веб-приложений Ruby on Rails, но с тех пор была существенно расширена и превратилась в самостоятельную библиотеку.
- Yahoo! User Interface (YUI): (*developer.yahoo.com/yui*) Библиотека элементов графического интерфейса, лучшая из недавних разработок компании Yahoo!, продолжающая свое развитие и предназначенная для использования сообществом.

В примере нашего приложения мы будем использовать библиотеку jQuery. Однако следует признать, что во многом этот выбор обусловлен личными предпочтениями авторов. Простейшие функциональные возможности технологии Ajax, которые мы будем использовать, обеспечиваются всеми библиотеками, представленными выше.

Структура каталогов приложения

Пришло время засучить рукава и приступить к работе! Пример этого приложения, которое мы назвали `liveupdate`, находится внутри проекта `liveproject`. Помимо приложения в каталоге проекта у нас имеется стандартная папка `templates` и папка `media` (где будут находиться сценарии JavaScript). Таким образом, первоначальная структура каталогов выглядит, как показано ниже (вывод получен с помощью команды `tree` в системе UNIX):

```
liveproject/
|-- __init__.py
|-- liveupdate
|   |-- __init__.py
|   |-- models.py
|   |-- urls.py
|   `-- views.py
|-- manage.py
|-- media
|   `-- js
|-- settings.py
|-- templates
|   `-- liveupdate
`-- urls.py
```

Обратите внимание, что структура папки `media` выбрана исключительно в соответствии с нашими собственными предпочтениями – платформа Django никак не регламентирует организацию дополнительных файлов и даже допускает их размещение за пределами папки проекта. В нашем случае мы руководствуемся привычками, приобретенными в процессе разработки крупных сайтов с большим количеством файлов JavaScript, CSS и изображений, когда наличие отдельной папки `js` имеет определенный смысл. В данном примере мы не используем внешние

файлы CSS или изображения, но если бы они имелись, мы создали бы для них папки img и css.

Размещение дополнительных файлов внутри каталога проекта упрощает управление этими файлами на сервере. Создание символической ссылки на каталог media в корневом каталоге документов веб-сервера Apache (и установка параметра настройки AllowSymlinks для веб-сервера Apache) гарантирует корректное обслуживание этих файлов.

Наше приложение liveupdate организовано в виде обычного пакета, включающего в себя все наши модели, определения адресов URL и представления. В настоящий момент, учитывая изложенные выше требования, определения адресов URL и модели отличаются удивительной простотой, а, кроме того, в приложении используются только универсальные представления. Ниже приводится содержимое файла URLconf liveupdate/urls.py уровня приложения (который должен подключаться с помощью функции include в файле urls.py уровня проекта), просто представляющее список объектов Update:

```
from django.conf.urls.defaults import *
from liveproject.liveupdate.models import Update

urlpatterns = patterns('django.views.generic',
    url(r'^$', 'list_detail.object_list', {
        'queryset': Update.objects.all()
    }),
)
```

Рассмотрим теперь файл models.py, где определяется класс Update модели (включая порядок сортировки по умолчанию), а также настройки для использования в приложении администрирования:

```
from django.db import models
from django.contrib import admin

class Update(models.Model):
    timestamp = models.DateTimeField(auto_now_add=True)
    text = models.TextField()

    class Meta:
        ordering = ['-id']

    def __unicode__(self):
        return "[%s] %s" % (
            self.timestamp.strftime("%Y-%m-%d %H:%M:%S"),
            self.text
        )

admin.site.register(Update)
```

Наконец, ниже приводится наша первая версия шаблона (templates/update_list.html), являющегося изначально «статическим» представлением текущего состояния обновляемого списка, который видят наши пользователи, когда первый раз загружают страницу:

```
<html>
    <head>
        <title>Live Update</title>
        <style type="text/css">
            body {
                margin: 30px;
                font-family: sans-serif;
                background: #fff;
            }
            h1 { background: #ccf; padding: 20px; }
            div.update { width: 100%; padding: 5px; }
            div.even { background: #ddd; }
            div.timestamp { float: left; font-weight: bold; }
            div.text { float: left; padding-left: 10px; }
            div.clear { clear: both; height: 1px; }
        </style>
    </head>
    <body>
        <h1>Welcome to the Live Update!</h1>
        <p>This site will automatically refresh itself every minute with new
        content - please <b>do not</b> reload the page!</p>

        {% if object_list %}
            <div id="update-holder">
                {% for object in object_list %}
               <div class="update {% cycle even,odd %}" id="{{ object.id }}">
                    <div class="timestamp">
                        {{ object.timestamp|date:"Y-m-d H:i:s" }}
                    </div>
                    <div class="text">
                        {{ object.text|linebreaksbr }}
                    </div>
                    <div class="clear"></div>
                </div>
                {% endfor %}
            </div>
        {% else %}
            <p>No updates yet - please check back later!</p>
        {% endif %}
    </body>
</html>
```

Как видите, с точки зрения управляющей логики, шаблон достаточно прост и в нем отсутствует какой-либо программный код JavaScript, так же как и отсутствуют инструкции подключения сценариев JavaScript. В следующем разделе мы дополним шаблон динамическими аспектами поведения, используя библиотеку jQuery.

Но прежде чем сделать это, давайте попробуем быстро протестировать наше приложение, чтобы получить представление об основных функциональных возможностях. Активируйте приложение администрирования (обращайтесь к предыдущим главам, если вам потребуется

вспомнить какие-то особенности), запустите команду `manage.py syncdb`, перезапустите веб-сервер Apache и откройте в броузере страницу администратора.

Вы должны увидеть обычную страницу управления нашей моделью `Update`. Щелкните на ссылке Add (добавить) и заполните область ввода текста, как показано на рис. 9.1. Обратите внимание: из-за того, что в определении поля timestamp мы использовали аргумент `auto_now_add`, нам необходимо ввести только текст, что обеспечивает возможность быстрого обновления содержимого нашего блога.

После добавления новой записи в административной странице появится вновь добавленный элемент `Update`, как показано на рис. 9.2.

Вы можете также увидеть «статическую» версию блога, как показано на рис. 9.3, введя в адресную строку адрес URL главной страницы.

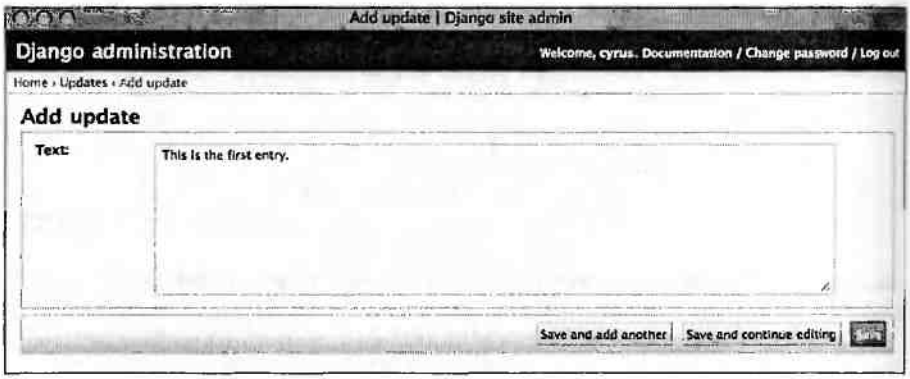

Рис. 9.1. Добавление новой записи Update

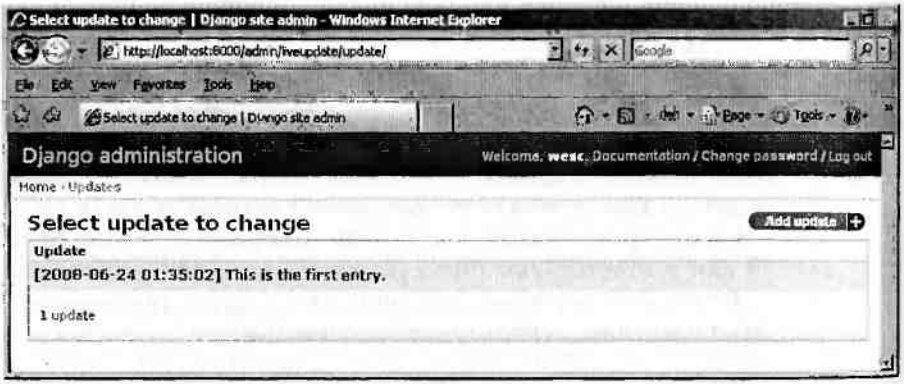

Рис. 9.2. Страница администрирования приложения Liveupdate после регистрации

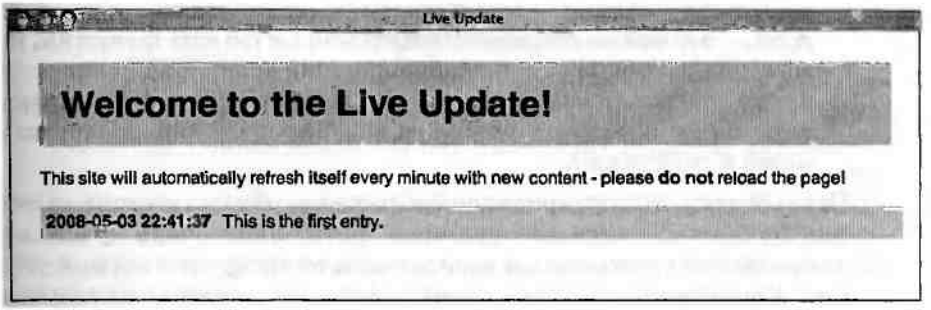

Рис. 9.3. Блог с единственной записью

Основа нашего приложения создана, теперь пришло время внедрить в него технологию Ajax.

Внедрение технологии Ajax

Этот раздел занимает большую часть главы, но не отчаивайтесь! Вам необходимо иметь некоторые теоретические знания, чтобы полностью понимать, что происходит, когда задействуется технология Ajax; поэтому мы, как обычно, даем вам возможность получить требуемые представления вместо простого копирования программного кода.

Для начала мы коротко познакомимся с тем, что необходимо для применения технологии Ajax и какие форматы она использует для передачи информации. Затем мы расскажем, как установить и проверить библиотеку JavaScript поддержки технологии Ajax. И в заключение покажем фактический программный код, который все это магическим образом реализует на стороне сервера и в броузере.

Основы

С практической точки зрения, «внедрение технологии Ajax» в веб-приложение делится на три основных этапа.

- Импортирование библиотеки: Так как для выполнения основной работы мы будем применять стороннюю библиотеку, нам необходимо включить ее в наш шаблон, чтобы получить возможность ее использовать.
- Определение функций обратного вызова на стороне клиента: Мы используем библиотеку для написания функции, которая выполняет мини-запрос к серверу и обновляет содержимое веб-страницы в соответствии с полученными результатами.
- Определение управляющей логики на стороне сервера: В конце концов, сервер должен знать, как отвечать на мини-запросы, поэтому мы должны определить функцию представления, которая будет

решать эту задачу – представления, совместимые с технологией Ajax, – это самые обычные представления (то есть функции, принимающие запрос HTTP и возвращающие ответ HTTP), которые попутно устанавливают один или два дополнительных заголовка в ответе HTTP (смотрите раздел «Создание функции представления» ниже в этой главе).

Первый этап, импортирование библиотеки, обычно состоит из включения JavaScript, хотя две функции[1] могут быть очень простыми или чрезвычайно сложными, в зависимости от требуемой логики управления. Как правило, большая часть работы выполняется на стороне клиента, так как серверная часть часто является всего лишь мостиком, связывающим JavaScript и базу данных; хотя это во многом зависит от ваших потребностей и от потребностей ваших пользователей. Кроме того, если вы заинтересованы в использовании аспектов библиотеки Ajax, связанных с созданием графического интерфейса, то этот программный код, который будет (или не будет) выполняться одновременно с мини-запросами, придется также поместить в шаблоны.

Символ «X» в аббревиатуре Ajax (или XML и JSON)

Диалог между клиентской и серверной функциями технически может осуществляться в любом формате, который возможно обработать в программном коде JavaScript (и/или в формате, который возможно передавать с помощью протокола HTTP, потому что именно этот протокол используется для поддержания диалога). Однако из-за того, что чаще всего бывает необходимо преобразовать или добавить в веб-страницу фрагмент разметки HTML, в большинстве случаев данные передаются в формате XML (где XHTML – одна из его разновидностей) или в текстовом формате JSON (JavaScript Object Notation – формат записи объектов JavaScript), представляющем собой обычный текст, который может быть преобразован в переменную JavaScript.

Формат XML, пожалуй, применяется чаще, и, несомненно, первая буква его названия используется в названии технологии Ajax из-за высокой популярности XML как языка межсистемного обмена данными. Учитывая близкие родственные отношения XML и (X)HTML, он также хорошо подходит для решения нашей задачи, потому что JavaScript и различные средства разработки веб-приложений уже предусматривают возможность манипулирования иерархическими структурами данных. Кроме того, разметку HTML можно полностью сформировать на стороне сервера (с применением шаблонов Django и механизма отображения), что позволит существенно упростить программный код функции на стороне клиента, которая просто будет встраивать полученный фрагмент в нужное место.

---

[1] По одной на стороне сервера и на стороне клиента. – *Прим. перев.*

JSON и Python

Во многих случаях данные в формате JSON могут быть преобразованы непосредственно в структуры данных на языке Python, что очень удобно, когда клиент выполняет передачу данных обратно на сервер. Однако существует несколько несовместимостей, таких как логические значения `true` и `false` в формате JSON и в языке Python (в котором первые символы этих значений записываются в верхнем регистре) или значение `null` в JSON и значение `None` в Python. В таких случаях необходимо использовать парсер Python/JSON. Подробнее о JSON и о возможности совместной работы с программным кодом на языке Python вы можете узнать на указанном выше веб-сайте JSON, а также в статьях *http://deron.meranda.us/python/comparing_json_modules/* и *http://blog.hill-street.net/?p=7*.

Не так давно начал набирать популярность формат JSON – благодаря краткому и удобочитаемому синтаксису, а также благодаря его большей компактности, чем XML. Плюс к этому синтаксис JSON близко напоминает определение структур данных на языке Python (строки, словари и списки). Подробнее о формате JSON вы можете узнать на сайте *http://json.org*.

Ниже приводится простой пример структуры данных в формате JSON.

```
{"first": "Bob", "last": "Smith", "favorite_numbers": [3,7,15]}
```

Даже если вы не знакомы с языком JavaScript, ваши познания в языке Python позволят вам понять, что это определение словаря, в котором два значения являются строками, а третье – это список целых чисел. Такая строка может играть роль структуры данных в JavaScript и использоваться программным кодом на стороне клиента.

В нашем приложении мы будем использовать формат JSON, хотя можно было бы выбрать и XML. Вам еще встретится множество примеров, описаний и руководств применения обоих форматов – как в печатном, так и в электронном виде.

Установка библиотеки JavaScript

Поскольку мы решили использовать в этом примере библиотеку jQuery, нам необходимо загрузить ее, воспользовавшись ссылкой Download на сайте *http://jquery.com*. jQuery распространяется в виде единой библиотеки и не делится на составляющие ее компоненты, как некоторые другие, но на сайте вы можете найти для загрузки различные варианты дистрибутивов – минимизированные, упакованные и неупакованные. Все три варианта обладают одинаковыми функциональными

возможностями, но имеют различные размеры и потребляют различное время процессора для распаковки.

Мы будем использовать минимизированный вариант текущей версии jQuery. К моменту написания этой книги текущей была версия 1.2.6, однако подойдет любая версия из серии 1.2.x. (Более старые версии также будут работать, но из-за отсутствия в них функции getJSON необходимо будет написать дополнительные инструкции для преобразования формата JSON в строки.)

Библиотеку jQuery необходимо поместить рядом с нашими собственными сценариями JavaScript, в каталоге liveproject/media/js, подключить к нашим шаблонам. В системе Win32 достаточно просто загрузить файл библиотеки в требуемый каталог. В UNIX-подобных системах, таких как Mac OS X и Linux, загрузку файлов можно выполнить с помощью инструментов командной строки wget или curl, как показано ниже (скопировав адрес URL загружаемого файла в броузере):

```
user@example:/opt/code/liveproject $ cd media/js/
user@example:/opt/code/liveproject/media/js $ wget
http://jqueryjs.googlecode.com/files/jquery-1.2.6.min.js
-2008-05-01 21:52:15 - http://jqueryjs.googlecode.com/files/jquery-1.2.6.min.js
Resolving jqueryjs.googlecode.com... 64.233.187.82
Connecting to jqueryjs.googlecode.com|64.233.187.82|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 54075 (53K) [text/x-c]
Saving to: 'jquery-1.2.6.min.js'

100%[=====================================================>] 54,075
227K/s in 0.2s

2008-05-01 21:52:16 (227 KB/s) - 'jquery-1.2.6.min.js' saved [54075/54075]
```

Теперь, когда библиотека находится в каталоге с дополнительными файлами, нам необходимо добавить ее в наш шаблон, включив следующую строку в тег <head> (обратите внимание, что /media/custom — это символическая ссылка, находящаяся в корневом каталоге документов веб-сервера Apache и указывающая на наш каталог liveproject/media):

```
<script type="text/javascript" language="javascript"
    src="/media/custom/js/jquery-1.2.6.min.js"></script>
```

Теперь все готово к использованию библиотеки jQuery, но сначала мы выполним простую настройку и проведем тестирование и только потом перейдем к реализации функциональных возможностей.

Настройка и тестирование библиотеки jQuery

Библиотеки поддержки технологии Ajax обычно обеспечивают простой способ доступа к программному коду, к текущему документу или к текущему объекту DOM. Библиотека jQuery использует уникальный синтаксис для доступа практически ко всем своим функциональным

возможностям – переменная $ привязана к специальному вызываемому объекту, который может вызываться как функция (например, $(argument)) или использоваться как объект, обладающий специальными методами (например, $.get(argument)).

Например, вызов $(document) возвращает объект, который представляет собой аналог переменной document в языке JavaScript, но снабженный некоторыми особенностями, характерными для jQuery. Одним из таких дополнительных методов является метод ready, используемый для вызова функции JavaScript после полной загрузки страницы (чтобы избежать проблем, свойственных функции onLoad в JavaScript).

В качестве примера использования этого метода (а также с целью подготовки к добавлению наших функциональных возможностей) добавьте в шаблон следующие строки в тег <head>, сразу же вслед за строкой, подключающей библиотеку jQuery:

```
<script type="text/javascript" language="javascript">
    $(document).ready(function() {
        alert("Hello world!");
    })
</script>
```

Язык JavaScript напоминает язык Python в том смысле, что функции в нем являются самыми обычными объектами – профессиональные разработчики широко используют возможность модификации объектов и передачу функций в виде аргументов. В данном случае вызов метода $(document).ready принимает функцию, которая выполняется после того, как страница будет полностью загружена, а, кроме того, мы объявляем ее как анонимную функцию – так же, как мы бы объявили лямбда-функцию Python, если бы ее можно было размещать в нескольких строках. Если все идет как надо и шаблон смог подключить библиотеку jQuery, наш фрагмент JavaScript просто будет выводить диалоговое окно со строкой «Hello World!» при обновлении страницы, что является прекрасным доказательством успешности испытания, хотя результат и не выглядит слишком захватывающим.

Внедрение операций JavaScript в наш шаблон

Попробуем реализовать более интересное для нас динамическое добавление элемента (представляющего объект Update, пусть даже и жестко заданный) в наш список во время загрузки страницы. Безусловно, это та же самая операция, которая впоследствии будет выполняться для добавления вновь поступающих данных.

```
<script type="text/javascript" language="javascript">
    $(document).ready(function() {
        $("#update-holder").prepend('<div class="update">\
                <div class="timestamp">2008-05-03 22:41:40</div>\
                <div class="text">Testing!</div>\
                <div class="clear"></div>\
```

```
            </div>');
    })
</script>
```

Здесь мы использовали синтаксис селекторов jQuery, который позволяет передавать функции `$()` строки, напоминающие селекторы CSS, и получать объект `Query`, представляющий все объекты, соответствующие этому селектору. В данном случае мы стремились отыскать тег `<div id="update-holder">`, определенный в предыдущем шаблоне, поэтому мы использовали символ `#`, который в языке CSS используется для выбора объектов по значению атрибута `id`.

После выбора требуемого объекта мы используем метод `prepend`, который добавляет объект или строку с разметкой HTML в начало выбранного объекта, то есть в данном случае он добавляет новый элемент `<div class="update">` в начало списка. Совсем неплохо для одной строки JavaScript! Взгляните, что получилось, когда мы загрузили страницу (рис. 9.4).

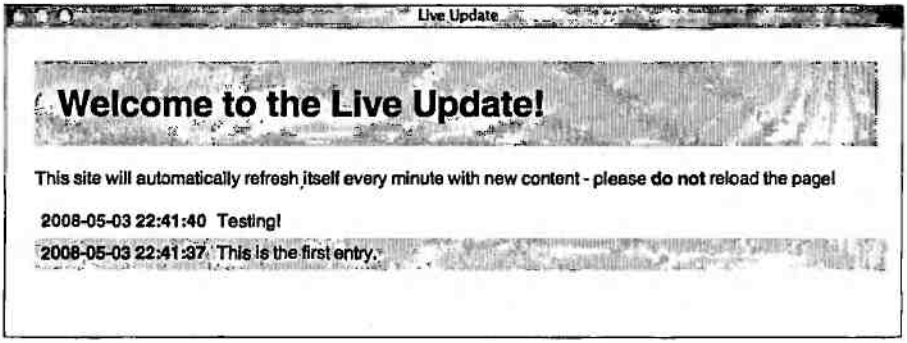

Рис. 9.4. Тестирование динамического обновления страницы с помощью JavaScript

Теперь мы точно знаем, что все настроено и работает. Теперь можно вернуться обратно на сервер и создать функцию представления. Мы реализуем тонкий интерфейс доступа к данным, который будет взаимодействовать с механизмом запросов библиотеки jQuery.

Создание функции представления

Самый простой и быстрый способ воплотить наши требования – запросить из JavaScript самые последние обновления, более новые, чем последнее сообщение, имеющееся в настоящий момент, – состоит в том, чтобы указать правильный адрес URL. Если бы нам потребовалось передавать на сервер более одного элемента данных, мы наверняка использовали бы запросы типа POST, но в данном случае мы можем реализовать все немного проще.

Временные метки и числовые идентификаторы

В действительности существует два способа упорядочения наших объектов Update: по числовому идентификатору и по времени. Использование числовых идентификаторов выглядит немного проще (так как при этом во время запросов не приходится преобразовывать строки с отметками времени (timestamp) в объекты datetime) и обеспечивает большую точность (в зависимости от частоты поступления объектов Update в базу данных может появиться более одного объекта с одним и тем же временем поступления!). Однако при использовании числовых идентификаторов предполагается, что их значения увеличиваются автоматически, как это автоматически происходит с соответствующими переменными в платформе Django, а в реальных обстоятельствах такое положение вещей обеспечивается не всегда.

Конечно, знание технических тонкостей – это только часть работы разработчика. Более сложной задачей часто оказывается понять, как воплотить предъявляемые требования в работоспособное решение.

Предположим, что наше представление поддержки технологии Ajax обслуживает адрес URL /updates-after/<id>/, где <id> – это идентификационный номер последнего объекта, доступного программному коду JavaScript. На основе этого идентификационного номера наше представление может выполнить простой запрос и вернуть все объекты Update, более новые, чем указанный в запросе. Данные, возвращаемые в формате JSON, представляют версии объектов модели, которые легко могут быть преобразованы сценарием JavaScript в разметку HTML.

Для этого нам необходимо добавить одну строку в файл URLconf уровня приложения, liveupdate/urls.py.

```
url(r'^updates-after/(?P<id>\d+)/$',
    'liveproject.liveupdate.views.updates_after'),
```

Ниже приводится соответствующая функция представления, находящаяся в файле liveupdate/views.py.

```
from django.http import HttpResponse
from django.core import serializers

from liveproject.liveupdate.models import Update

def updates_after(request, id):
    response = HttpResponse()
    response['Content-Type'] = "text/javascript"
    response.write(serializers.serialize("json",
```

```
        Update.objects.filter(pk__gt=id)))
    return response
```

Мы сэкономили массу времени, использовав библиотеку сериализации, встроенную в платформу Django, которая способна преобразовывать объекты модели в произвольные текстовые форматы, включая XML, JSON и YAML. Функция `serializers.serialize` выполняет преобразование объектов, которые запрос `QuerySet` отбирает по первичному ключу (`pk`) – в нашем случае мы запрашиваем только те объекты, значения числовых идентификаторов которых больше указанного значения `id`.

Функция возвращает строку в формате по нашему выбору, в данном случае – JSON, которую мы записываем в ответ, используя поведение, похожее на работу с файлом, объекта `HttpResponse`. В заключение в ответе устанавливается заголовок `Content-Type`, который необходим, чтобы обеспечить корректную интерпретацию и использование тела ответа в JavaScript.

Вся прелесть удобочитаемых текстовых форматов заключается в том, что они легко поддаются отладке. На рис. 9.5 показано, что произойдет, если попытаться вручную открыть в броузере страницу с адресом URL *http://localhost:8000/updates-after/0/* при наличии единственного тестового объекта `Update` в базе данных.

Мы практически закончили наше приложение, осталось лишь на основе всего вышеизложенного написать программный код JavaScript, который свяжет страницу с этим представлеием и будет обновлять страницу при получении ответа.

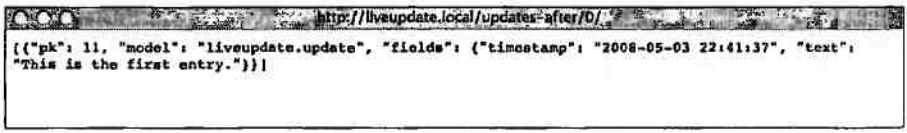

Рис. 9.5. *Тестирование представления, возвращающего данные в формате JSON, в броузере*

Использование функции представления в JavaScript

В языке JavaScript имеется встроенная функция таймера, способная выполнять произвольный программный код через определенные интервалы времени. Эта функция называется `setInterval`, она принимает строку, которую требуется интерпретировать, и период срабатывания таймера в миллисекундах, например:

```
setInterval("update()", 60000);
```

Эта строка обеспечит вызов функции update() каждые 60 000 миллисекунд, или 60 секунд. Добавив использование удобного метода getJSON из библиотеки jQuery, который производит мини-запрос по указанному адресу URL и интерпретирует ответ как строку в формате JSON, мы наконец добрались до использования технологии Ajax в нашей странице. Ниже приводится окончательный результат:

```
<script type="text/javascript" language="javascript">
    function update() {
        update_holder = $("#update-holder");
        most_recent = update_holder.find("div:first");
        $.getJSON("/updates-after/" + most_recent.attr('id') + "/",
            function(data) {
                cycle_class = most_recent.hasClass("odd")
                    ? "even" : "odd";
                jQuery.each(data, function() {
                    update_holder.prepend('<div id="' + this.pk
                        + '" class="update ' + cycle_class
                        + '"><div class="timestamp">'
                        + this.fields.timestamp
                        + '</div><div class="text">'
                        + this.fields.text
                        + '</div><div class="clear"></div></div>'
                    );
                    cycle_class = (cycle_class == "odd")
                        ? "even" : "odd";
                });
            }
        );
    }
    $(document).ready(function() {
        setInterval("update()", 60000);
    })
</script>
```

Действия, выполняемые внутри функции update, должны быть достаточно очевидны, тем не менее, ниже приводится краткое их описание:

1. С помощью различных методов отбора, входящих в состав библиотеки jQuery, функция отыскивает контейнерный объект (update_holder) и самый последний элемент Update (most_recent).
2. В первом аргументе метода getJSON конструируется срока URL с участием атрибута ID элемента HTML most_recent (в качестве значения которого, для удобства, на стороне сервера был установлен идентификационный номер, полученный из базы данных).
3. Второй аргумент метода – это обычная анонимная функция, реализация которой описывается в остальных пунктах.
4. Первая строка анонимной функции инициализирует переменную класса CSS значением «even» или «odd».

5. Затем с помощью функции each из библиотеки jQuery выполняется обход данных в формате JSON, полученных от функции представления. Эти данные представлены в виде списка сериализованных объектов Update.
6. Эти объекты Update используются для конструирования новых фрагментов разметки HTML, которые затем добавляются в начало контейнерного элемента .
7. В конце каждой итерации циклически изменяется класс CSS, чтобы обеспечить вывод четных и нечетных строк разным цветом.

Теперь, когда функция update определена, оказывается, что весь программный код, фактически выполняемый внутри функции ready, – это вызов упоминавшейся ранее функции setInterval. После добавления очередной записи в блог она должна автоматически загрузиться в веб-страницу – в течение минуты с момента сохранения записи в базе данных. Мы не можем продемонстрировать этот программный код в действии, так как анимированные изображения GIF или видеоролики невозможно представить в книге, тем не менее, на рис. 9.6 приводится снимок с экрана, демонстрирующий, как выглядит страница сайта после того, как было добавлено несколько записей.

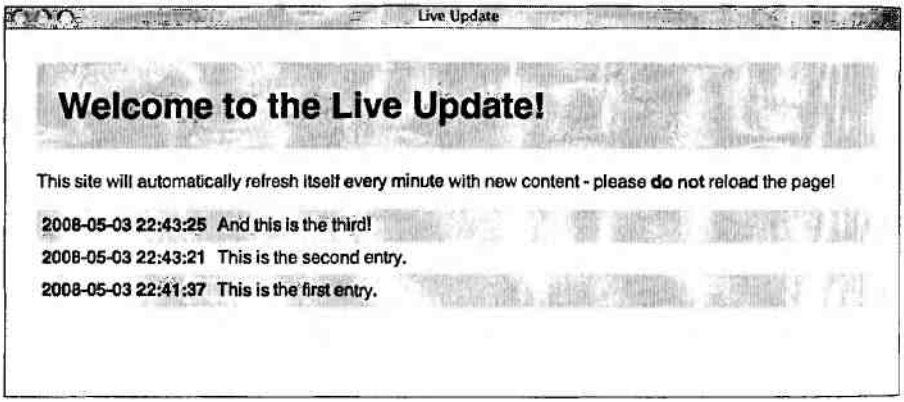

Рис. 9.6. Окончательный вид страницы блога с несколькими записями

В заключение

Хотя для реализации функциональных возможностей на стороне клиента мы использовали мощный синтаксис селекторов библиотеки jQuery и ее функцию getJSON, необходимо отметить, что ни то ни другое было совершенно необязательно для того, чтобы воспользоваться функциональностью на стороне сервера, реализованной в этом примере. Для работы с нашей маленькой функцией представления достаточ-

но легко можно было бы обойтись вообще без использования какой-либо библиотеки поддержки технологии Ajax.

Ключевым здесь является организация взаимодействий по протоколу HTTP – наша функция представления ожидает получить обычный запрос GET HTTP, а похожие функции представления могли бы использовать метод POST, и возвращает значение (отправляемое по протоколу HTTP) также в открытом формате, JSON. Как внутренние компоненты Django основаны на идее максимальной гибкости и независимости, так и использование технологии Ajax вместе с платформой Django ориентировано на использование открытых и хорошо структурированных решений.

В этой главе основное внимание было уделено стороне Ajax, связанной с мини-запросами, однако существует масса интересных материалов для чтения, касающихся удивительных возможностей применения JavaScript для создания пользовательского интерфейса. Мы рекомендуем вам ознакомиться с приложением D «Поиск, оценка и использование приложений на платформе Django», где приводятся несколько адресов, откуда можно будет начать, но помните – весьма заманчиво поместить в свое приложение массу графических «рюшек и бантиков», но ваши пользователи будут вам благодарны, только если вы проявите в этом чувство меры!

10 Зак. 1496

10

Pastebin

Самым привлекательным в платформе Django для многих разработчиков является то, что она позволяет писать веб-приложения на языке Python. Но, как это ни парадоксально, другой привлекательной чертой Django являются ее универсальные представления, дающие возможность создавать веб-приложения, написав совсем немного программного кода на языке Python. (С универсальными представлениями вы познакомились в главе 5 «Адреса URL, механизмы HTTP представления»).

Универсальные представления могут быть мощным инструментом быстрой разработки и макетирования. Но пусть вас не вводит в заблуждение такое их название. Универсальные представления – это не просто временные «подпорки». Начинающим пользователям Django можно простить то, что они видят за этим названием предоставление некоторого шаблона по умолчанию, используемого для отображения данных, но ведь на самом деле реалии совсем не таковы. Не забывайте, что **представление** в платформе Django – это программный код на языке Python, который принимает объект `HttpRequest` запроса HTTP и возвращает объект `HttpResponse` ответа. Разработка структуры объектов данных, передаваемых представлению, и шаблона, используемого для отображения ответа, полностью находятся в вашем ведении.

В этом разделе мы пройдем через процесс создания простого приложения Pastebin[1], которое опирается на использование универсальных

---

[1] Из Википедии: «pastebin, или nopaste, – веб-приложение, которое позволяет загружать отрывки текста, обычно фрагменты исходного кода, для возможности просмотра окружающими. Такой сервис очень популярен среди пользователей IRC сетей, где вставка больших фрагментов текста считается плохим тоном». – *Прим. перев.*

представлений. Ниже приводится перечень особенностей нашего приложения:

- Форма для передачи новых элементов с одним обязательным (содержимое) и двумя необязательными полями (имя отправителя и заголовок)
- Список последних элементов, щелчком на которых можно выполнять переход к просмотру этих элементов
- Представление для просмотра каждого элемента
- Интерфейс администратора, позволяющий нам (как владельцам сайта) редактировать или удалять существующие записи
- Подсветка синтаксиса
- Периодическое удаление устаревших записей

Для достижения описанных выше функциональных возможностей нам практически не придется писать программный код на языке Python, реализующий управляющую логику. Мы создадим файл модели, описывающей наши данные и их атрибуты и шаблоны отображения этих данных, но практически все остальные аспекты приложения будут реализованы за счет использования встроенных механизмов платформы Django.

Примечание

Программный код этого приложения по своей функциональности напоминает первую версию сайта *dpaste.com* – приложения Pastebin сообщества Django. В настоящее время *dpaste.com* уже не является веб-приложением, опирающимся исключительно на универсальные представления, но сущность его простоты и практичности присутствует в программном коде этой главы.

Одно небольшое примечание, прежде чем мы двинемся дальше: этот пример наглядно демонстрирует, какой значительный объем работы мы можем передать платформе. Некоторые могли бы назвать такой подход подходом для ленивых, но каждая строка программного кода, которую вы не напишете, не потребует отладки в дальнейшем. Поэтому в данном примере везде, где будет возможность выбирать между написанием небольшого объема программного кода, чтобы реализовать некоторые специфические особенности поведения, и возможностью перепоручить работу платформе Django, мы позволим платформе самой выполнять работу.

Определение модели

Ниже приводится полное содержимое файла `models.py` для нашего приложения Pastebin. В нем определяется простая структура данных, состоящая из пяти полей, некоторый параметр в классе `Meta` и пара методов, используемых программным кодом представления и нашими

шаблонами. В нем также выполняется регистрация модели в приложении администрирования и устанавливаются некоторые параметры администрирования, имеющие отношение к отображению списка.

```
import datetime
from django.db import models
from django.db.models import permalink
from django.contrib import admin

class Paste(models.Model):
    """Структура единственной записи в приложении Pastebin"""

    SYNTAX_CHOICES = (
        (0, "Plain"),
        (1, "Python"),
        (2, "HTML"),
        (3, "SQL"),
        (4, "Javascript"),
        (5, "CSS"),
    )

    content = models.TextField()
    title = models.CharField(blank=True, max_length=30)
    syntax = models.IntegerField(max_length=30, choices=SYNTAX_CHOICES,
                                 default=0)
    poster = models.CharField(blank=True, max_length=30)
    timestamp = models.DateTimeField(auto_now_add=True, blank=True)

    class Meta:
        ordering = ('-timestamp',)

    def __unicode__(self):
        return "%s (%s)" % (self.title or "#%s" % self.id,
                            ➥self.get_syntax_display())

    @permalink
    def get_absolute_url(self):
        return ('django.views.generic.list_detail.object_detail',
            None, {'object_id': self.id})

class PasteAdmin(admin.ModelAdmin):
    list_display = ('__unicode__', 'title', 'poster', 'syntax', 'timestamp')
    list_filter = ('timestamp', 'syntax')

admin.site.register(Paste, PasteAdmin)
```

Большинство этих элементов мы уже видели ранее. Отличие здесь только в строках, содержащих наш собственный программный код на языке Python для данного приложения. За исключением нескольких простых правил в файле urls.py основная работа передана самой платформе.

Приложения, опирающиеся на универсальные приложения, такие как в данном примере, на практике демонстрируют мощь принципа

«не повторяйся». В примере приложения, которое мы собираемся создавать, пять полей, составляющих основу описанной выше модели (content, title, syntax, poster и timestamp), будут использоваться:

- Командой manage.py syncdb для создания таблицы в базе данных
- Приложением администрирования для воссоздания интерфейса редактирования наших данных
- Универсальным представлением object_detail, которое будет извлекать экземпляры модели и передавать их системе шаблонов для отображения
- Универсальным представлением create_update, которое генерирует и обрабатывает форму добавления нового элемента

Методы модели также будут использоваться в нескольких местах. Приложение администрирования будет использовать метод __unicode__ объекта, когда потребуется сослаться на объект по имени (например, в сообщениях, требующих подтверждения выполнения операции удаления), и метод get_absolute_url – для создания ссылок на отдельные записи. Шаблоны будут неявно использовать метод __unicode__ везде, где потребуется отобразить название элемента, а метод get_absolute_url – для создания ссылок в списке последних поступлений.

Создание шаблонов

Теперь создадим несколько простых шаблонов, которые будут использоваться для отображения содержимого сайта. Сначала создадим базовый шаблон, этот прием вы уже видели в главе 7 «Фотогалерея». Сохраните следующие строки в файле pastebin/templates/base.html.

```
<html>
    <head>
        <title>{% block title %}{% endblock %}</title>
        <style type="text/css">
            body { margin: 30px; font-family: sans-serif; background: #fff; }
            h1 { background: #ccf; padding: 20px; }
            pre { padding: 20px; background: #ddd; }
        </style>
    </head>
    <body>
        <p><a href="/paste/add/">Add one</a> &bull;
        <a href="/paste/">List all</a></p>
        {% block content %}{% endblock %}
    </body>
</html>
```

После создания базового шаблона перейдем к созданию формы, с помощью которой пользователи смогут добавлять свой программный код в наше приложение. Сохраните следующие строки в файле pastebin/

templates/pastebin/paste_form.html. **(Кажущаяся избыточность в пути к файлу объясняется ниже.)**

```
{% extends "base.html" %}
{% block title %}Add{% endblock %}
{% block content %}
<h1>Paste something</h1>
<form action="" method="POST">
Title: {{ form.title }}<br>
Poster: {{ form.poster }}<br>
Syntax: {{ form.syntax }}<br>
{{ form.content }}<br>
<input type="submit" name="submit" value="Paste" id="submit">
</form>
{% endblock %}
```

Затем создадим шаблон списка. В этом списке будут отображаться последние добавленные элементы, и пользователям будет предоставляться возможность переходить к ним щелчком мыши. Сохраните следующие строки в файле pastebin/templates/pastebin/paste_list.html.

```
{% extends "base.html" %}
{% block title %}Recently Pasted{% endblock %}
{% block content %}
<h1>Recently Pasted</h1>
<ul>
    {% for object in object_list %}
    <li><a href="{{ object.get_absolute_url }}">{{ object }}</a></li>
    {% endfor %}
</ul>
{% endblock %}
```

В заключение создадим шаблон просмотра одного элемента. Это шаблон, на просмотр которого люди будут тратить больше всего времени. Сохраните следующие строки в файле pastebin/templates/pastebin/paste_detail.html.

```
{% extends "base.html" %}
{% block title %}{{ object }}{% endblock %}
{% block content %}
<h1>{{ object }}</h1>
<p>Syntax: {{ object.get_syntax_display }}<br>
Date: {{ object.timestamp|date: "r" }}</p>
<code><pre>{{ object.content }}</pre></code>
{% endblock %}
```

Определение адресов URL

Структура нашего приложения настолько прозрачна, что определения адресов URL выглядят чрезвычайно просто. Единственная хитрость заключается в том, чтобы задействовать универсальные представле-

ния. Нам необходимо определить три шаблона адресов URL: один – для списка всех элементов, один – для отображения отдельных элементов и один – для страницы добавления нового элемента.

```
from django.conf.urls.defaults import *
from django.views.generic.list_detail import object_list, object_detail
from django.views.generic.create_update import create_object
from pastebin.models import Paste

display_info = {'queryset': Paste.objects.all()}
create_info = {'model': Paste}

urlpatterns = patterns('',
    url(r'^$', object_list, dict(display_info, allow_empty=True)),
    url(r'^(?P<object_id>\d+)/$', object_detail, display_info),
    url(r'^add/$', create_object, create_info),
)
```

В действительности эти строки составляют основу нашего приложения. Здесь из пакета `django.view.generic` импортируются три функции представлений:

- `django.views.generic.list_detail.object_list`
- `django.views.generic.list_detail.object_detail`
- `django.views.generic.create_update.create_object`

В дополнение к объекту `HttpRequest`, который все представления Django, как универсальные, так и любые другие, принимают в первом аргументе, этим представлениям также передаются словари с дополнительными значениями. В файле URLconf выше мы определили два различных словаря. Имена `display_info` и `create_info` были выбраны произвольно (хотя суффикс `_info` часто употребляется в именах таких словарей), но их содержимое структурировано с учетом особенностей используемых универсальных представлений. Представление `list_detail` ожидает получить в словаре ключ `queryset`, значением которого является объект `QuerySet`, содержащий все элементы, удовлетворяющие критериям. Представления из модуля `create_update` ожидают получить класс (а не экземпляр) модели в виде значения ключа `model`. Для представления `object_list` мы добавили в словарь ключ `allow_empty=True`, чтобы указать представлению, что страница должна отображаться, даже если в базе данных нет ни одного объекта.

Существует еще множество других значений, которые можно включить в эти словари. Так как они представляют собой основное средство настройки поведения универсальных представлений, они могут иметь довольно существенный объем. Полный перечень параметров, принимаемых этими представлениями, вы найдете в официальной документации к платформе Django. А пока постараемся сохранить максимальную простоту.

Примечание

Существует специальное правило, касающееся использования словарей `_info`, которое вам следует знать. Программный код в файле URLconf не интерпретируется заново при каждом запросе. Это означает, что если не предпринять специальных действий, информация, хранящаяся в объекте `Paste.objects.all`, фактически может устареть по мере добавления, редактирования или удаления записей пользователями или администраторами сайта. К счастью, платформа Django знает об этом и потому ей явно предписывается не кэшировать данные, поставляемые в виде значения ключа `queryset`.

Запуск приложения

Хотя мы написали совсем немного программного кода, у нас теперь имеется достаточно функциональное приложение. Давайте опробуем его. После запуска приложения мы увидели страницу, как показано на рис. 10.1 – пустой список записей.

Отвлечемся на минуту и представим, что потребовалось сделать платформе Django, чтобы получить эту страницу: она проанализировала запрошенный адрес URL; определила, какое представление требуется вызвать; передала (пустой) объект `QuerySet`, полученный из нашей модели; отыскала файл шаблона; отобразила шаблон, используя соответствующее содержимое; и вернула сгенерированную страницу с разметкой HTML броузеру.

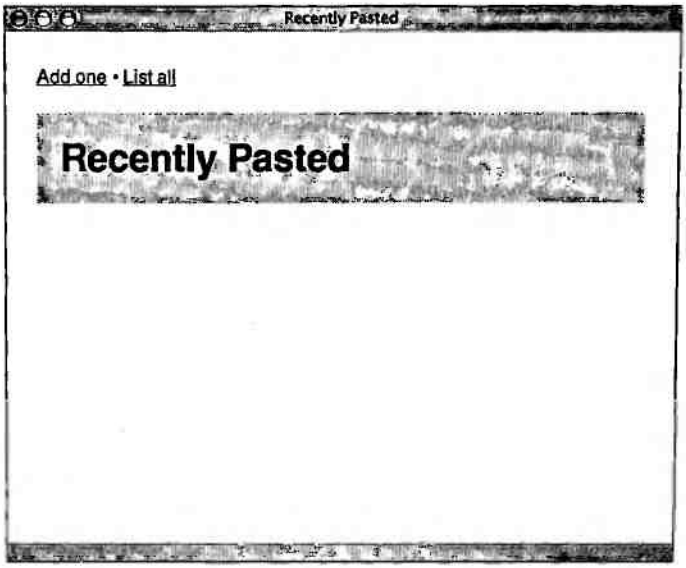

Рис. 10.1. Пустой список записей

Теперь добавим некоторое содержимое. Если щелкнуть на ссылке Add One (добавить элемент), в окне броузера должна появиться пустая форма. Эта форма является результатом сотрудничества функции представления Django и нашего шаблона. Универсальное представление create_update исследует структуру модели, генерирует элементы формы HTML и передает их нашему шаблону в виде переменной шаблона {{ form }}. Наш шаблон распаковывает полученные элементы и отображает форму в окне броузера. Обратите внимание, что ответственность за добавление тега <form> и кнопки отправки формы целиком возлагается на нас.

В окончательном виде форма должна выглядеть, как показано на рис. 10.2.

Наше дружественное приложение Pastebin не перегружает пользователя работой. Фактически достаточно будет заполнить лишь поле Code (код). Такие приложения, как Pastebin, будут приносить практическую пользу, только если они удобны в обращении, а длинные списки полей, обязательных к заполнению, – это очень неудобно.

Заполнив форму и щелкнув на кнопке Paste (вставить), мы снова вызываем универсальное представление create_update платформы Django. Так как теперь данные передаются методом POST, а не GET, представление понимает, что вместо отображения пустой формы необходимо обработать ввод пользователя и (если это возможно) сохранить его в базе данных.

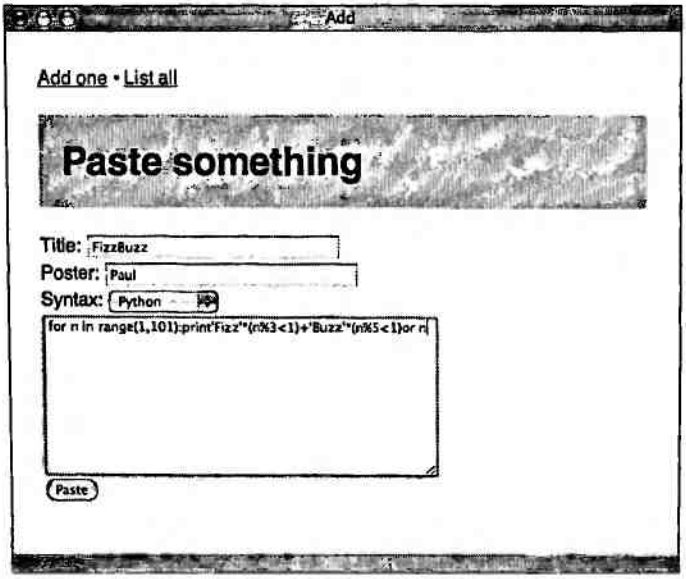

Рис. 10.2. Форма добавления одной записи

Одной из особенностей представления `create_update`, которую мы не рассматривали здесь, является механизм проверки корректности введенных данных. Что произойдет, если пользователь оставит обязательное поле незаполненным? В этом случае форма будет выведена повторно. В нашем чрезвычайно простом примере отсутствует программный код, который принимал бы и отображал сообщения об ошибках, но в действительности платформа Django *передает* их в виде переменной шаблона `{{ form.errors }}`. Более надежная реализация могла бы принимать эти сообщения об ошибках и отображать их для обеспечения большего удобства.

Предположим, что пользователь заполнил единственное обязательное поле и щелкнул на кнопке Paste (вставить), а платформа Django обработала введенные данные (с помощью все того же универсального представления `create_update`) и сохранила их в базе. После этого она перенаправит пользователя к вновь созданному объекту, по адресу `get_absolute_url`, и отобразит отправленные данные с помощью шаблона `pastebin_detail.html`, как показано на рис. 10.3.

Наиболее ожидаемое поведение, на которое рассчитывает пользователь после отправки новой записи, – представление этой записи и адрес URL, который он сможет отправить другим.

Если пользователь пожелает узнать, какие еще записи присутствуют в приложении Pastebin, ссылка List All (вывести все) даст ему такую возможность. Все имеющиеся записи будут выводиться в виде просто-

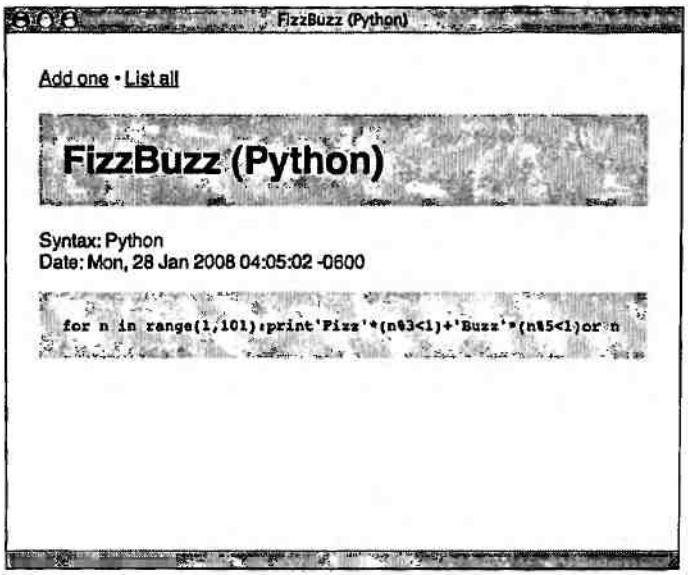

Рис. 10.3. Вновь добавленная запись

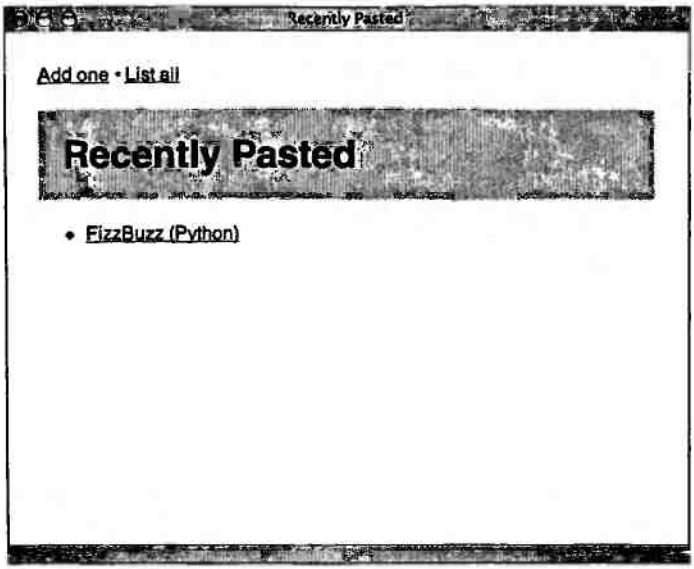

Рис. 10.4. Список отправленных записей

го маркированного списка, как показано на рис. 10.4, с помощью универсального представления `object_list` и шаблона `paste_list.html`.

Этот список прекрасно иллюстрирует работу универсального представления `object_list`. С помощью словаря `display_info`, который определен в нашем файле URLconf, в представление `object_list` передается объект `QuerySet` со всеми нашими объектами `Paste`. Это представление передает объекты нашему шаблону, где цикл `{% for object in object_list %}` превращает их в ссылки.

Примечание

С практической точки зрения, список всех записей необязательно является самой значимой особенностью сайтов данного типа. Пользователи приложения Pastebin обычно проявляют интерес только к своим записям. Владельцы приложений типа Pastebin часто задаются вопросом: «Почему спамеры все время что-то добавляют в мое приложение?». Ответ прост: спамеров интересуют любые механизмы, позволяющие показать свою информацию ничего подобного не ожидающим пользователям. Список последних поступлений в Pastebin прекрасно справляется с поставленной задачей, при этом оставаясь неудобным для размещения коммерческих объявлений.

Наконец, не забывайте, что сверх всего того, что мы создали более или менее явно, мы получаем еще и приложение администрирования. Учитывая параметры, которые были определены в классе `PasteAdmin`, административный раздел нашего сайта выглядит, как показано на рис. 10.5.

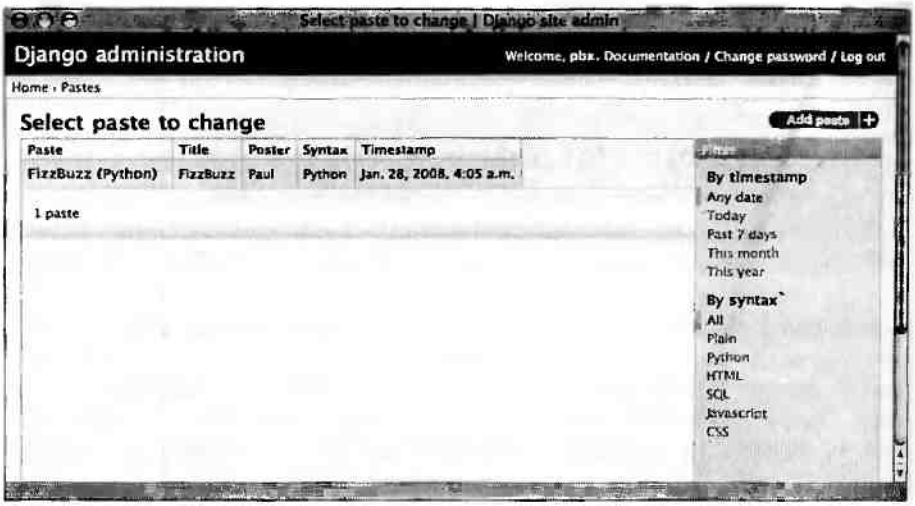

Рис. 10.5. Страница администратора

Обратите внимание, что первым аргументом методу `list_display` класса `PasteAdmin` передается не какое-то определенное поле модели, а метод `__unicode__`, благодаря чему создаются ссылки, текст которых зависит от того, какая информация доступна.

Это полезная возможность для приложений, полностью опирающихся на использование универсальных представлений платформы Django. Хотя было бы вполне разумно начинать расширение возможностей приложения с создания специализированных функций представлений, тем не менее, множество улучшений можно внести, продолжая использовать универсальные представления. Мы внесем три улучшения: добавим возможность управления списком последних поступлений, подсветку синтаксиса и автоматическое удаление устаревших записей.

Ограничение числа записей в списке последних поступлений

Список последних поступлений выглядит прекрасно при небольшом количестве записей, но если сайт имеет высокую посещаемость, такой список может очень быстро стать слишком громоздким. Существуют разные способы ограничить число записей в этом списке. Учитывая, что мы используем только универсальные представления, лучшим местом, где можно было бы реализовать эти ограничения, является шаблон.

Для этого достаточно изменить шестую строку шаблона `paste_list.html`, применив фильтр `slice` к объекту `object_list`.

```
{% for object in object_list|slice:":10" %}
```

Вот как это работает: файл URLconf передает в шаблон объект QuerySet, представляющий все записи в базе данных. Они уже сортированы в порядке отдаления времени добавления – благодаря наличию строки ordering = ('-timestamp') в файле models.py. Цикл for шаблона извлекает первые десять записей и выполняет итерации по ним.

Значение, передаваемое фильтру slice, в точности соответствует значению, которое мы использовали бы в обычной операции извлечения среза на языке Python, за исключением скобок. Эквивалентный пример на языке Python можно представить примерно так:

```
>>> number_list = [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> print number_list[:10]
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6]
```

Если бы объекты QuerySet не откладывали выполнение операций с базой данных, то есть, если бы шаблону передавался полный список объектов только для того, чтобы выбросить все записи, кроме первых десяти, это было бы безумием. Если бы у нас имелись тысячи записей, потребление памяти процессом веб-сервера взлетело бы до предела. Благодаря тому, что объекты QuerySet откладывают выполнение операций с базой данных до самого последнего момента (в данном случае таким последним моментом является цикл for в нашем шаблоне), мы без всякого ущерба можем указать в файле URLconf вызов метода Paste.objects.all() и затем получать срез в шаблоне. Кроме того, поскольку число, отражающее количество элементов в списке, относится к отображению, шаблон является отличным местом для реализации такого ограничения.

Подсветка синтаксиса

Приложение Pastebin будет намного полезнее (и привлекательнее), если оно будет знать, как применять подсветку синтаксиса к добавляемым фрагментам. Существуют различные способы реализации этой возможности (включая замечательную библиотеку **Pygments** на языке Python, которая используется сайтом *dpaste.com*), но самый простой способ заключается в том, чтобы реализовать подсветку синтаксиса на языке JavaScript на стороне клиента.

Программист Алекс Горбачев (Alex Gorbatchev) создал на языке JavaScript отличную реализацию подсветки синтаксиса. Называется эта утилита Syntax Highlighter, а получить ее можно на сайте Google Code (*http://code.google.com/p/syntaxhighlighter/*).

Полный комплект инструкций и примеров использования Syntax Highlighter можно найти на веб-сайте проекта, а здесь мы лишь покажем, как добавить подсветку синтаксиса для фрагментов программного кода на языке Python.

Сначала дополним шаблон paste_detail.html, который теперь выглядит, как показано ниже:

```
{% extends "base.html" %}
{% block title %}{{ object }}{% endblock %}
{% block content %}
<h1>{{ object }}</h1>
<p>Syntax: {{ object.get_syntax_display }}<br>
Date: {{ object.timestamp|date:"r" }}</p>
<code><pre name="code" class="{{ object.get_syntax_display|lower }}">
    {{ object.content }}</pre></code>
<link type="text/css" rel="stylesheet"
    href="/static/css/SyntaxHighlighter.css"></link>
<script language="javascript" src="/static/js/shCore.js"></script>
<script language="javascript" src="/static/js/shBrushPython.js"></script>
<script language="javascript" src="/static/js/shBrushXml.js"></script>
<script language="javascript" src="/static/js/shBrushJscript.js"></script>
<script language="javascript" src="/static/js/shBrushSql.js"></script>
<script language="javascript" src="/static/js/shBrushCss.js"></script>
<script language="javascript">
    dp.SyntaxHighlighter.HighlightAll('code');
</script>
{% endblock %}
```

Мы добавили в тег <pre> атрибуты name и class. Это позволяет программному коду JavaScript отыскивать наши фрагменты.

```
<pre name="code" class="{{ object.get_syntax_display|lower }}">
```

Это все, что потребовалось. Когда броузер будет отображать страницу, он выполнит программный код JavaScript, реализующий подсветку синтаксиса, который преобразит черно-белые невыразительные фрагменты программного кода, прежде чем пользователь увидит их. Результат должен выглядеть примерно так, как показано на рис. 10.6.

Удаление устаревших записей с помощью задания cron

Записи, отправляемые на сайты Pastebin, обычно недолго сохраняют свою актуальность, поэтому будет нелишним организовать периодическую проверку и автоматическое удаление устаревших записей. Лучше всего реализовать такую очистку в виде **задания cron**, запускаемого каждую ночь на вашем сервере.

Задания cron и прочие сценарии Django, которые предназначены для работы вне окружения веб-сервера, – это еще одно свидетельство практической ценности принципа, утверждающего, что Django – «это всего лишь программный код на языке Python». При создании сценариев, имеющих дело с объектами приложений на платформе Django, при-

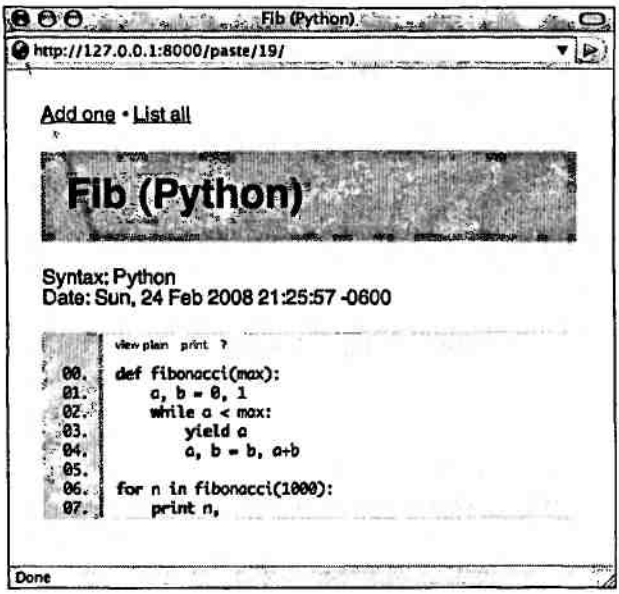

Рис. 10.6. Фрагмент программного кода на языке Python с подсветкой синтаксиса

влекается не так много особенностей, специфичных для Django. Наш простой сценарий зависит от следующих предположений:

- Переменная окружения `DJANGO_SETTINGS_MODULE` содержит строку, определяющую на языке Python путь к файлу с настройками проекта (например, «pastesite.settings»).
- В модуле `settings` проекта имеется параметр `EXPIRY_DAYS`.
- Проект называется «pastesite».

Предполагая, что об удалении устаревших записей мы позаботились, останется позаботиться только о тестировании и развертывании приложения.

```
#!/usr/bin/env python
import datetime
from django.conf import settings
from pastesite.pastebin.models import Paste

today = datetime.date.today()
cutoff = (today - datetime.timedelta(days=settings.EXPIRY_DAYS))
Paste.objects.filter(timestamp__lt=cutoff).delete()
```

Основная работа выполняется в последней строке сценария – она использует механизм ORM Django для выбора всех объектов в базе данных, добавленных раньше расчетного времени, и удаляет их все сразу.

Примечание

В зависимости от используемого механизма баз данных, можно также «вычищать», или освобождать пространство, занимаемое удаленными записями. Простейший вариант реализации такой очистки для базы данных SQLite можно найти по адресу: *http://djangosnippets.org*.

В заключение

Хотелось бы надеяться, что теперь вы убедились в широких возможностях универсальных представлений Django. Наш пример сайта Pastebin оказался достаточно прост в реализации, но только представьте все особенности, которыми он обладает: проверка вводимых данных, перенаправление после отправки формы, возможность просмотра списка и отдельных записей и т. д. Но, пожалуй, еще лучше, чем наличие всех этих особенностей, – это знание, что благодаря использованию платформы Django мы имеем надежную основу для внесений расширений. Если нам потребуется внести в наше приложение некоторые ограничения или уменьшить объем внутреннего трафика, добавив кэширование, то платформа Django всегда у нас под рукой.

IV

Дополнительные возможности и особенности Django

11. Передовые приемы программирования в Django
12. Передовые приемы развертывания Django

11

Передовые приемы программирования в Django

В этой главе мы исследуем множество различных профессиональных приемов, которые можно использовать при программировании приложений на платформе Django; среди них такие, как распространение информации по каналам RSS и в других форматах, настройка поведения приложения администрирования и использование расширенных возможностей системы шаблонов.

В следующей главе «Передовые приемы развертывания Django» рассматриваются похожие темы, такие как тестирование, импортирование данных и создание сценариев обслуживания, напрямую не относящиеся к основной логике приложения. Для обеих глав порядок чтения разделов не имеет большого значения – вы можете перескакивать между разделами, как сочтете нужным.

Настройка приложения администрирования

Приложение администрирования платформы Django в определенной степени можно считать драгоценным камнем в короне. Может показаться странным, что мы решили поговорить о приложении, расположенном в каталоге «contrib», который больше подходит для размещения вспомогательных, необязательных дополнений, но никак не базовых компонентов. Однако разработчики Django пометили приложение администрирования так, исходя из следующего соображения: вы *не обязаны* использовать его при работе с платформой Django, поэтому оно отнесено к разряду необязательных. Приложение администрирование – это мощное приложение, привлекательное своими возможностями. Если вам требуется быстро создать удобный в использовании

интерфейс ввода и редактирования данных, приложение администрирования станет вашим верным другом.

На протяжении всей книги вам приходилось видеть, как выполняется настройка приложения с помощью подклассов `ModelAdmin`. Например, параметры `list_display`, `list_filter` и `search_fields` обеспечивают большую часть базовых настроек, которые могут потребоваться.

Кроме того, упоминалось, что административный раздел, используемый в наших примерах, является административным разделом «по умолчанию», то есть имеется в виду, что существует возможность создать несколько административных разделов для обеспечения большей гибкости. Это позволяет нам организовывать различные административные представления, например для различных групп пользователей.

Однако со временем неизбежно появляется желание получить от приложения администрирования еще больше возможностей. Для начинающих разработчиков приложений на платформе Django после того, как они попользуются приложением администрирования некоторое время, вполне типично высказывание: «Есть у меня одна задумка, но, прочитав документацию, я так и не смог выяснить, как ее реализовать. Вот если бы мне *удалось воплотить свой замысел*, это был бы идеальный инструмент!»

Далее мы продемонстрируем вам некоторые способы настройки и расширения приложения администрирования на примере реализации некоторых особенностей, о которых часто спрашивают. Однако, раз уж вы читаете этот раздел, имейте в виду, что *приложение администрирования – это всего лишь еще одно приложение на платформе Django*. Чрезвычайно удобно иметь приложение с привлекательным интерфейсом и с возможностью настройки в широких пределах, но, в конечном счете, если оно не способно делать то, что вам требуется, в вашей власти заменить его чем-то другим.

В зависимости от того, насколько глубоко вам приходится погружаться в настройки, может оказаться гораздо дальновиднее создать свое приложение администрирования, чем пытаться настроить существующее под задачи, для решения которых оно не было предназначено. Если широко известные приемы, предлагаемые здесь, не удовлетворяют ваших потребностей, то, вероятно, для вас настало время подумать о создании собственного приложения администрирования. После этого короткого отступления перейдем к рассмотрению некоторых расширенных возможностей настройки приложения администрирования.

Изменение расположения и стилей элементов с помощью параметра fieldsets

Параметр `fielfsets` настройки приложения администрирования обеспечивает полный контроль над отображением данных. Например, есть вероятность, что у вас возникнет желание использовать специальные

стили CSS для определенных элементов, группировать поля каким-то определенным способом, добавлять к выбранным полям вспомогательный программный код JavaScript или указывать, что некоторые поля должны находиться в скрытом состоянии.

Ниже приводится тривиальный пример модели, в которой настройки отображения полей выполнены с помощью параметра fieldsets.

```
class Person(models.Model):
    firstname = models.CharField(max_length=50)
    lastname = models.CharField(max_length=50)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=2)

class PersonAdmin(admin.ModelAdmin):
    fieldsets = [
        ("Name", {"fields": ("firstname", "lastname")}),
        ("Location", {"fields": ("city", "state")})
    ]

admin.site.register(Person, PersonAdmin)
```

Выражаясь на языке Python, параметр fieldsets – это список **кортежей, состоящих из двух элементов.** Первый элемент каждого кортежа – это строковая метка группы полей, а второй элемент – словарь параметров настройки этой группы. Ключами словаря являются имена определенных параметров (они будут перечислены ниже), значениями которых могут быть данные различных типов, в зависимости от конкретного параметра. В нашем простом примере параметр fields определяет кортеж имен полей.

В результате использования параметра fieldsets в административном интерфейсе поля будут сгруппированы, как показано на рис. 11.1.

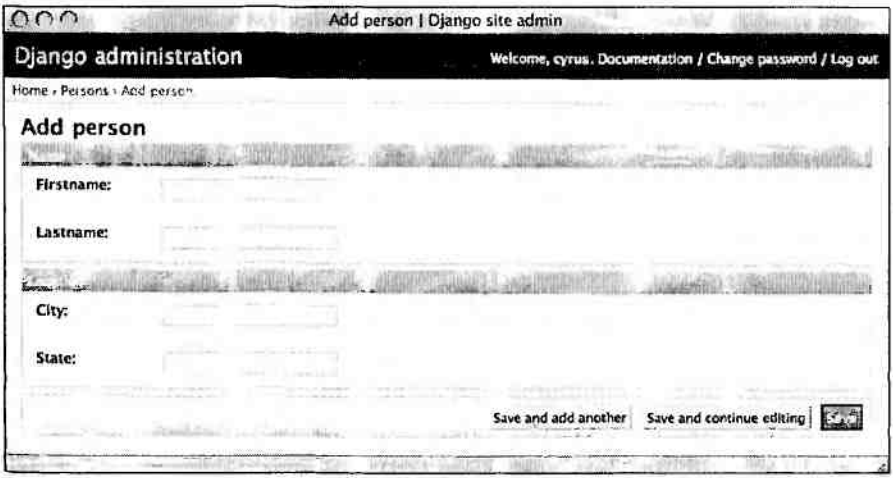

Рис. 11.1. Новые поля в административном интерфейсе

Ниже перечислены параметры, которые можно использовать (в качестве ключей словаря из предыдущего примера) внутри параметра настройки fieldsets.

- classes: Этот параметр определяет кортеж строк с именами классов CSS (доступных в шаблоне отображения), которые должны применяться к этой группе полей. В приложении администрирования имеется несколько предопределенных классов, которые можно использовать здесь: «collapsed» – заставляет группу полей свернуться в заголовок, ее можно переключать из программного кода JavaScript между свернутым и развернутым состояниями; «monospace» – может применяться к полям textarea, которые будут содержать фрагменты программного кода; и «wide» – обеспечивает увеличение ширины (хотя и фиксированное) полей в приложении администрирования. Загляните в подкаталог «media», находящийся в каталоге django.contrib.admin, где вы найдете каскадные таблицы стилей по умолчанию, если вы хотите узнать, какие еще классы можно использовать.

Примечание

Если вы практикуете стиль «прогрессивного улучшения» при разработке на JavaScript, классы CSS, добавляемые с помощью этого параметра, могут играть роль механизмов запуска вашего собственного программного кода JavaScript – например, если вы решите добавить редактор WYSIWYG в элемент textarea.

- description: Этот параметр определяет строку, которая будет использоваться как описание группы полей. Его можно рассматривать как атрибут help_text группы полей. Стили CSS, используемые приложением администрирования по умолчанию, написаны так, чтобы отображать такой текст удобным и ненавязчивым способом. Подобные приятные мелочи дизайна придают еще больше значимости таким простым особенностям маркировки, как эта.
- fields: Как уже говорилось выше, этот параметр определяет кортеж с именами полей, которые визуально должны объединяться в группу при отображении в административном интерфейсе. Если словарь, в котором присутствует данный параметр, предваряется строкой (например, «Name» в предыдущем фрагменте), она используется в качестве текстовой метки для группы полей. В противном случае группа будет отображаться без текстового заголовка.

Расширение базовых шаблонов

Помимо простого изменения параметров настройки имеется также возможность произвести более существенные настройки внешнего вида приложения администрирования, заменив один или более базовых шаблонов своими версиями. Конечно, в действительности не требуется *заменять* сами шаблоны, достаточно лишь переопределить их своими файлами; как это сделать – рассказывается ниже.

С технической точки зрения, приложение администрирования – это всего лишь еще одно приложение на платформе Django, хотя и более сложное. Сложная иерархия шаблонов, используемых этим приложением, мало пригодна для изучения начинающими программистами, не имеющими должной подготовки. Для начала познакомимся с шаблоном, который называется base_site.html. Ниже приводится полное его содержимое, немного переформатированное для удобства представления.

```
{% extends "admin/base.html" %}
{% load i18n %}
{% block title %}{{ title|escape }}|{% trans 'Django site admin' %}
{% endblock %}
{% block branding %}
<h1 id="site-name">{% trans 'Django administration' %}</h1>
{% endblock %}
{% block nav-global %}{% endblock %}
```

Будучи знакомыми с системой шаблонов Django, вы уже можете предположить, что тег extends, находящийся в самом верху, вероятно, выполняет основную работу, и это действительно так. Более сложный шаблон называется base.html и представляет собой основу всех страниц приложения администрирования. Однако в шаблоне base_site.html имеются два элемента, которые вы, вероятно, пожелали бы настроить, – достаточно абстрактное содержимое тега <title> и заголовка <h1>.

Чтобы выполнить эти настройки, просто замените строки «Django site admin» и «Django administration» на более подходящие для вашего приложения и сохраните шаблон. Но где его сохранить?

Чтобы переопределить содержимое шаблона, необходимо создать собственную измененную копию и затем помочь загрузчику шаблонов Django обнаружить его раньше шаблона по умолчанию. Вы можете сохранить свою копию либо внутри каталога с шаблонами приложения (если вы используете загрузчик шаблонов app_directories), либо в одном из каталогов, перечисленных в параметре настройки TEMPLATE_DIR.

В любом случае, вам необходимо поместить шаблон в подкаталог admin, чтобы теги extends в других шаблонах приложения администрирования смогли отыскать его.

Если вы помещаете шаблон внутри своего приложения (например, myproject/myapp/templates/admin/base_site.html), в параметре INSTALLED_APPS обязательно укажите свое приложение до django.contrib.admin. Загрузчик шаблонов выполняет поиск в каталогах в соответствии с указанным порядком следования приложений, поэтому, если приложение администрирования будет указано перед вашим приложением, загрузчик шаблонов Django первым обнаружит оригинальный шаблон и прекратит дальнейший поиск.

Если вам потребуется выполнить более глубокие настройки шаблонов приложения администрирования, то следующий шаблон, на который следует обратить внимание, – это `index.html`. Он немного более сложный, чем `base_site.html`, но его стоит исследовать, если появится желание более существенно изменить главную страницу приложения администрирования.

Добавление новых представлений

Не желая создавать полностью свое приложение администрирования, существенную настройку административного раздела можно выполнить посредством создания собственных представлений. Это наиболее предпочтительный способ непосредственного изменения программного кода приложения администрирования.

Однако с каждым обновлением основной версии Django сопровождение ваших изменений будет ложиться тяжким грузом на ваши плечи. Если в основную версию Django будут вноситься изменения, вступающие в конфликт с вашей реализацией, вам придется постоянно заниматься их объединением. С другой стороны, если ваши настройки принимают форму самостоятельных представлений, их легко можно будет передавать другим разработчикам, пользующимся платформой Django, которые могут передавать вам свои исправления и улучшения.

Создание новых представлений – это лишь вопрос использования платформы Django, так как в приложении администрирования не слишком много специфических особенностей. Существуют три основных требования, предъявляемых к представлениям:

- В идеале ваше представление должно отображаться на адрес URL, находящийся «внутри» пространства адресов URL приложения администрирования. Это обеспечит большее удобство для пользователя, чем для вас, однако система URLconf платформы Django позволяет легко сделать это, поэтому нет уважительного повода не сделать этого.
- Представление должно генерировать страницу, по своему внешнему виду напоминающую остальные страницы административного раздела. Говоря другими словами, представление должно выводить свои ответы с помощью шаблонов приложения администрирования.
- Для ограничения доступа представление должно использовать декораторы аутентификации, входящие в состав платформы Django, о которых рассказывается в следующем разделе.

Декораторы аутентификации

Если вы плохо знакомы с языком Python и вам необходимо вспомнить, что представляют собой декораторы, то, прежде чем продолжить чтение этого раздела, дочитайте этот параграф, а затем вернитесь

к разделу с описанием декораторов в главе 1 «Практическое введение в Python для Django». Декораторы – это функции, которые изменяют другие функции. Представления в платформе Django – это тоже функции. Их необходимо изменить так, чтобы только определенные пользователи могли вызывать их, а такие изменения легко реализуются с помощью декораторов.

Декоратор можно представить себе как щит для функции представления. Если пользователь прошел вашу проверку, вызывается функция представления, которая возвращает страницу для отображения. Если пользователь не прошел проверку, он перенаправляется в другое место. (По умолчанию пользователь перенаправляется по адресу /accounts/login/ с дополнительным параметром, указывающим, куда должен быть перенаправлен пользователь в случае успешной регистрации, например /accounts/login/?next=/jobs/101/. Этот адрес URL определяется параметром LOGIN_URL в файле settings.py.)

Для обеспечения принудительной аутентификации достаточно использовать единственный декоратор, который называется user_passes_test.

Чтобы иметь возможность использовать этот декоратор, помимо функции представления, доступ к которой необходимо защитить, вам также потребуется предоставить вспомогательную функцию, реализующую фактическую проверку. Эта функция должна принимать объект User и возвращать логическое значение (True или False). Она может быть очень простой и даже тривиальной, как показано ниже:

```
def user_is_staff(user):
    return user.is_staff
```

При наличии такого определения функции порядок использования декоратора может выглядеть примерно так, как показано ниже:

```
@user_passes_test(user_is_staff)
def for_staff_eyes_only(request):
   print "Next secret staff meeting date: November 14th"
```

Если такое определение функции user_if_staff кажется вам излишним, можно использовать лямбда-функцию (краткое введение в лямбда-функции вы найдете в главе 1). С помощью ключевого слова lambda мы можем определить анонимную функцию прямо в вызове функции-декоратора.

```
@user_passes_test(lambda u: u.is_staff)
def for_staff_eyes_only(request):
    ...
```

Развивая наш пример дальше, допустим, что нам необходимо более тонкое разделение прав, чем простое определение сотрудник-или-несотрудник. Одна из удобных особенностей приложений администрирования и аутентификации платформы Django состоит в том, что они

реализуют удобную систему разрешений для управления пользователями. Вы уже знаете, что каждая модель получает в приложении администрирования свой собственный набор разрешений на выполнение операций создания/изменения/удаления, но эти разрешения также доступны в программном коде на языке Python. Если, к примеру, у вас имеется модель SecretMeeting, снабженная своими привилегиями в системе аутентификации, и представление, позволяющее отдельным пользователям назначать конфиденциальные встречи, то декорированная функция могла бы выглядеть примерно так:

```
@user_passes_test(lambda u: u.can_create_secretmeeting)
def secret_meeting_create(request):
    ...
```

Объединяя предопределенные разрешения со своими собственными, свойственными для вашего приложения, вы получаете возможность полностью управлять доступом, как вам требуется.

Приложение Syndication

В состав платформы Django входит приложение Syndication (django.contrib.syndication), представляющее собой удобное средство создания лент RSS или Atom из любых наборов объектов моделей. Оно позволяет удивительно легко добавлять ленты в проекты на платформе Django.

Приложение Syndication может настраиваться в очень широких пределах, но его легко начать использовать благодаря весьма разумным настройкам по умолчанию. Чтобы задействовать это приложение, необходимо выполнить всего два основных условия: определить специальный класс для создания объектов ленты и добавить в файл URLconf правило, которое будет передавать эти объекты приложению Syndication. Чтобы вам было проще понять, как все это работает, мы сразу же перейдем к программному коду. За основу этого примера было взято приложение блога из главы 2 «Django для нетерпеливых: создание блога».

Класс Feed

Чтобы настроить свои ленты, необходимо определить отдельный модуль, содержащий специальные классы, который затем будет импортироваться непосредственно в файл URLconf. Платформа Django не накладывает никаких ограничений на местоположение этого файла (как и на его имя), но разумнее все-таки поместить его в каталог приложения и дать ему имя feeds.py. Итак, в каталоге mysite/blog создайте файл с именем feeds.py, содержащий следующие строки:

```
from django.contrib.syndication.feeds import Feed
from mysite.blog.models import BlogPost
```

```
class RSSFeed(Feed):
    title = "My awesome blog feed"
    description = "The latest from my awesome blog"
    link = "/blog/"
    item_link = link

    def items(self):
        return BlogPost.objects.all()[:10]
```

Атрибуты title и description используются подписчиком (например, приложением чтения лент RSS) для маркировки ленты. Атрибут link указывает страницу, ассоциированную с лентой. Указанный вами адрес URL будет предваряться доменным именем сайта.

Атрибут item_link определяет, какую страницу загрузить, если пользователь пожелает просмотреть веб-страницу, ассоциированную с отдельным элементом ленты. Установив в атрибуте item_link адрес /blog/, например, мы тем самым определим, что щелчок на любом элементе ленты будет отправлять пользователя на начальную страницу нашего блога, но если вам кажется негибким такое решение, то вы правы. Лучше будет снабдить модель BlogPost методом get_absolute_url, который автоматически будет использоваться приложением Syndication.

Метод items является основой этого класса – он определяет, какие объекты следует вернуть приложению Syndication. В данном случае метод возвращает первые десять объектов из списка сообщений в блоге, а так как модель BlogPost имеет атрибут Meta.ordering со значением "-timestamp" (в обратном хронологическом порядке), мы получим их в обратном хронологическом порядке.

Созданный класс минимален ровно настолько, чтобы быть работоспособным. Он содержит только те элементы, которые совершенно необходимы приложению Syndication. Существует еще множество дополнительных атрибутов и множество гибких способов задания значений обязательных атрибутов. При исследовании возможных настроек важно понимать, что любой атрибут класса Feed в действительности относится к одному из трех типов:

- Жестко определенное значение (как атрибуты title, description и link в предыдущем примере).
- Метод, не имеющий явных аргументов (неявный аргумент self, ссылающийся на сам экземпляр класса, конечно же, является обязательным).
- Метод, явно принимающий один явный аргумент (отдельный элемент ленты).

Приложение Syndication автоматически определяет тип атрибута, опираясь на преимущества «утиной типизации» в языке Python. Приложение Syndication просматривает типы атрибутов в порядке, обратном указанному выше. Если обнаруживается, что атрибут имеет форму метода с одним аргументом, приложение вызывает этот метод,

передавая ему текущий объект. В противном случае проверяется, не является ли атрибут методом без аргументов, и если это так, – вызывает его. Если первые две проверки завершились неудачей, выполняется попытка получить жестко определенное значение. Если все три попытки потерпели неудачу, возбуждается исключение FeedDoesNotExist (поскольку в этом случае считается, что лента имеет ошибочное определение).

Определение адреса URL ленты

После создания класса, который фактически обслуживает создание ленты, все, что остается сделать, – это определить рабочий адрес URL ленты. Продолжая наш пример с блогом, мы могли бы дополнить файл URLconf приложения (mysite/blog/urls.py), как показано ниже:

```
from django.conf.urls.defaults import *
from django.contrib.syndication.views import feed
from mysite.blog.views import archive
from mysite.blog.feeds import RSSFeed

urlpatterns = patterns('',
    url(r'^$', archive),
    url(r'^feeds/(?P<url>.*)/$', feed, {'feed_dict': {'rss': RSSFeed}}),
)
```

Мы добавили ровно три строки. Две – с инструкциями import, которые импортируют представление feed из приложения Syndication и наш новый класс RSSFeed из приложения блога.

Мы также добавили достаточно сложный шаблон адреса URL. Разделив его на три составляющие, получаем:

- r'^feeds/(?P<url>.*)/$': Регулярное выражение адреса URL. Так как мы находимся внутри приложения «blog», которому соответствует адрес URL /blog/, то это регулярное выражение будет сохранять /blog/feeds/FOO/ в группу с именем «FOO», которая будет передаваться функции представления.
- feed: Функция представления, которая была импортирована из модуля django.contrib.syndication.views.
- {'feed_dict': {'rss': RSSFeed}}: В любом кортеже urlpatterns мы можем указать в третьем элементе словарь, который используется для передачи дополнительных аргументов в функцию представления. Здесь мы передаем один аргумент с именем feed_dict, который является словарем с единственным элементом, отображающим строку "rss" в класс RSSFeed. Мы могли бы добавить другие типы лент, просто создав необходимые классы и сославшись на них в этом словаре.

Несмотря на то, что в качестве ключа нашего контекстного словаря dict("rss") мы использовали название типа ленты, тем не менее, мы могли бы использовать все, что угодно, например "latest" для обозначения списка последних сообщений. Приложение Syndication не огра-

ничивает нас в выборе ключа. Ему достаточно знать, какие классы Feed оно может использовать для обработки запросов.

Дополнительные возможности работы с лентами

Организация передачи лент новостей RSS или Atom может оказаться существенным усовершенствованием для регулярно (или нерегулярно) обновляемого сайта. Несмотря на кажущуюся простоту этих форматов, при разработке собственного программного кода, осуществляющего вывод лент новостей, легко допустить ошибку. Широко известный парсер Universal Feed Parser (*http://feedparser.org/*) распространяется с более чем *тремя тысячами тестов*, позволяющими убедиться, что он надежно обрабатывает широкий диапазон (некоторые предпочитают вместо слова «диапазон» говорить «месиво») типов лент. Используя приложение Syndication, вы получаете ясный и корректный вывод за счет небольшого объема программного кода и настроек.

Если вам требуется выполнить более специфические настройки, которые не были описаны здесь, обращайтесь к официальной документации к платформе Django. Весьма маловероятно, что вам когда-нибудь придется писать свою собственную реализацию.

Создание загружаемых файлов

Так как Django является платформой разработки веб-приложений, было бы естественно воспринимать ее как инструмент создания разметки HTML и передачи ее по протоколу HTTP. Однако платформу Django легко приспособить для создания содержимого других типов и распространяемого другими способами.

Это возможно благодаря двум факторам. Первый заключается в том, что язык шаблонов имеет обычный текстовый формат, а не формат XML. Веб-платформа, которая позволяет создавать шаблоны только в формате XML, не в состоянии оказать существенную помощь в создании простых текстовых отчетов или сообщений электронной почты.

Второй фактор состоит в том, платформа Django обеспечивает доступ к механизмам HTTP, управляющим вашим сайтом, и позволяет определять различные заголовки HTTP. Благодаря этому можно определить в заголовке Content-Type тип содержимого, отличный от HTML, например JavaScript (как в представлении JSON в главе 9 «Живой блог»), или заголовок Content-Disposition, чтобы обеспечить принудительную загрузку документа вместо его отображения в броузере.

Ниже приводятся несколько коротких примеров использования Django для вывода документов, не являющихся веб-страницами. В некоторых из них используется система шаблонов, как уже упоминалось выше. В других – нет, потому что в определенных случаях *любая* система шаблонов – это просто ненужные накладные расходы. Как обычно,

старайтесь использовать тот инструмент, который лучше подходит для решения поставленной задачи.

Конфигурационные файлы Nagios

Один из примеров, которые мы коротко рассмотрим здесь, – это широко известная открытая система мониторинга Nagios (*http://nagios.org/*). Как и многие другие подобные проекты, система Nagios следует принятым в операционной системе UNIX соглашениям о распространении конфигурационных файлов в простом текстовом формате. Такие файлы представляют собой отличную мишень для системы шаблонов Django.

Один из авторов в процессе работы над этим разделом создал небольшое и легко настраиваемое приложение для внутреннего использования, которое (кроме всего прочего) выполняет частичную настройку Nagios. Оно опирается на приложение Django, которое играет роль центральной базы данных с информацией о системах и службах. Этот каталог экспортируется в формат Nagios, что дает пользователю возможность вести учет систем в одном месте и осуществлять их мониторинг с помощью системы Nagios.

Ниже приводится упрощенный пример иерархии моделей систем и служб (в действующем приложении эта информация распределена по большему количеству моделей).

```
class System(models.Model):
    name = models.CharField(max_length=100)
    ip = models.IPAddressField()

class Service(models.Model):
    name = models.CharField(max_length=100)
    system = models.ForeignKey(System)
    port = models.IntegerField()
```

Ниже следует шаблон, который генерирует файлы Nagios проверки служб для каждого хоста, – также сильно упрощенный по сравнению с действующими шаблонами. Контекстом шаблона является объект `system` типа `System`.

```
define host {
    use             generic-host
    host_name       {{ system.name }}
    address         {{ system.ip }}
}

{% for service in system.service_set.all %}
define service {
    use                     generic-service
    host_name               {{ system.name }}
    service_description     {{ service.name }}
    check_command           check_tcp!{{ service.port }}
}
```

Примечание

Пусть вас не вводят в заблуждение одиночные фигурные скобки – они являются частью формата файлов Nagios и не распознаются системой шаблонов Django. Система шаблонов опознает только двойные фигурные скобки и фигурные скобки в паре со знаками процента.

При отображении предыдущий шаблон дает нам работающий файл Nagios, где определяется формат Nagios хост-системы и выводится список всех служб, связанных с нею, с помощью сетевых команд проверки, которые проверяют, находится ли указанный порт TCP в использовании.

vCard

Формат vCard – это текстовый формат, используемый для представления контактной информации. Этот формат поддерживается многими популярными приложениями адресных книг, клиентов электронной почты и PIM (Personal Information Manager – менеджер личной информации), включая Evolution, OS X Address Book и Microsoft Outlook.

Если в вашем приложении на платформе Django хранится какая-либо контактная информация, может потребоваться ее экспорт в формат vCard, чтобы пользователи могли импортировать эту информацию в свои локальные адресные книги или приложения PIM.

Следующий фрагмент программного кода использует модуль vObject (доступный по адресу: *http://vobject.skyhouseconsulting.com/*), позволяющий упростить воспроизведение данных в формате vCard.

```
import vobject

def vcard(request, person):
    v = vobject.vCard()
    v.add('n')
    v.n.value = vobject.vcard.Name(family=person.lastname,
                                   given=person.firstname)
    v.add('fn')
    v.fn.value = "%s %s" % (person.firstname, person.lastname)
    v.add('email')
    v.email.value = person.email
    output = v.serialize()
    filename = "%s%s.vcf" % (person.firstname, person.lastname)
    response = HttpResponse(output, mimetype="text/x-vCard")
    response['Content-Disposition'] = 'attachment; filename=%s' % filename
    return response
```

Наиболее важным здесь является манипулирование объектом `HttpResponse`, в котором определяется тип MIME, отличный от HTML, и указывается имя файла для загрузки. Вместо того чтобы возвращать информацию непосредственно в объекте `HttpResponse`, мы сначала создаем

объект, определяем содержимое и тип MIME. Затем устанавливаем заголовок Content-Disposition и указываем, что ответ содержит вложение. Возвращая объект ответа, наша функция представления приводит механизм в действие, и пользователь получает сгенерированный файл.

Этот прием наглядно демонстрирует превосходное сочетание инструментов низкого и высокого уровней в платформе Django. Объект HttpResponse позволяет нам применять на практике наши знания о броузерах, серверах и о протоколе HTTP для настройки возвращаемого ответа, не задумываясь при этом о других составляющих ответа, которые нам не нужны.

Этот способ может использоваться для передачи данных любых типов, отличных от HTML, как будет показано в следующих примерах.

Значения, разделенные запятыми (CSV)

Когда появится необходимость извлекать из вашего приложения табличные данные, ничто не сможет сравниться с форматом CSV (comma-separated value – значения, разделенные запятыми). Работу с этим форматом можно реализовать на любом существующем под солнцем языке программирования, и практически любое приложение, предназначенное для работы со структурированными данными (Microsoft Excel, Filemaker Pro), может импортировать и экспортировать данные в этом формате. В языке Python имеется удобный модуль csv, который поможет вам в этом.

Самое первое, что пытаются сделать программисты, когда сталкиваются с необходимостью обработки данных в формате CSV, – это написать свой собственный парсер или генератор. Ведь формат CSV очень прост, разве не так? Всего лишь запятые и числа, ну и, возможно, некоторые другие символы. И, может быть, кавычки, чтобы окружать ими значения, содержащие запятые. И, может быть, экранирующие последовательности, если внутри кавычек имеются другие кавычки. Однако эта задача начинает сильно обрастать частностями! Приятно, что кто-то уже написал модуль, который заботится обо всем об этом.

Предположим, что нам необходимо представить данные, хранящиеся в объектах person из предыдущего примера работы с форматом vCard, в виде электронной таблицы с колонками для имени, фамилии и адреса электронной почты. Для начала воспользуемся интерактивной оболочкой интерпретатора, чтобы убедиться в корректной работе модуля csv. Так как модуль csv проектировался для работы с объектами, напоминающими файлы, для перехвата выводимой им информации будем использовать удобный модуль StringIO. (Модуль StringIO реализует интерфейс файлов для строк.)

```
>>> import csv, StringIO
>>> output = StringIO.StringIO()
>>> output_writer = csv.writer(output)
```

```
>>> people = [("Bob", "Dobbs", "bob@example.edu"),
... ("Pat", "Patson", "pat@example.org"),
... ("O,RLY", "O'Reilly", "orly@example.com")]
>>> for p in people:
...     output_writer.writerow(p)
...
>>> print output.getvalue()
Bob,Dobbs,bob@example.edu
Pat,Patson,pat@example.org
"O,RLY",O'Reilly,orly@example.com
```

Отлично, модуль заключил в кавычки проблемный элемент, содержащий запятую. А как будет выглядеть функция представления? Практически так же – благодаря тому, что объекты HttpResponse в платформе Django, как и объекты StringIO, обладают «файловым интерфейсом», то есть они имеют метод write, который требуется объекту csv.writer.

```
def csv_file(request, people):
    import csv
    response = HttpResponse(mimetype="text/csv")
    response_writer = csv.writer(response)
    for p in people:
        response_writer.writerow(p)
    response['Content-Disposition'] = 'attachment; filename=everybody.csv'
    return response
```

Когда пользователь отправляет запрос, он попадает в это представление, которое также возвращает объект HttpResponse – это общее требование для всех представлений Django – но вместо содержимого типа text/html он несет в себе содержимое типа text/csv, а также имеет заголовок Content-Disposition, который в большинстве броузеров вызывает операцию загрузки данных вместо их отображения.

Формат CSV дает положительный ответ на вопрос: «возможно ли импортировать данные из приложения на платформе Django в приложение X». Практически любая программа, предназначенная для работы со структурированными данными, способна читать данные в формате CSV.

Вывод диаграмм и графиков с помощью библиотеки PyCha

PyCha (*http://www.lorenzogi.com/projects/pycha/*) – это простая библиотека для языка Python, предназначенная для вывода диаграмм и графиков и основанная на использовании графической системы Cairo. Библиотека PyCha не стремится обеспечить вывод во всех возможных форматах и широкий выбор параметров настройки, но у нее имеются две сильные стороны: относительно «питонический» синтаксис и привлекательный внешний вид результатов с настройками по умолчанию.

Как и в предыдущем примере с форматом CSV, основная хитрость состоит в том, чтобы организовать вывод в строку, которую можно вернуть в ответе, и установить соответствующий тип MIME ответа.

```
def pie_chart(request):
    import sys, cairo, pycha.pie
    data = (
        ('Io on Eyedrops', 61),
        ('Haskell on Hovercrafts', 276),
        ('Lua on Linseed Oil', 99),
        ('Django', 1000),
    )
    dataset = [(item[0], [[0, item[1]]]) for item in data]
    surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, 750, 450)
    ticks = [dict(v=i, label=d[0]) for i, d in enumerate(data)]
    options = {
        'axis': {'x': {'ticks': ticks}},
        'legend': {'hide': True}
    }
    chart = pycha.pie.PieChart(surface, options)
    chart.addDataset(dataset)
    chart.render()
    response = HttpResponse(mimetime="image/png")
    surface.write_to_png(response)
    response['Content-Disposition'] = 'attachment; filename=pie.png'
    return response
```

Главные этапы здесь – это получение объекта surface, который является основным элементом рисования в системе Cairo, создание диаграммы посредством передачи этого объекта конструктору pycha.pie.PieChart и последующий вызов метода render объекта диаграммы. После отображения диаграммы вызывается метод write_to_png объекта surface, записывающий двоичные данные в формате PNG в объект, напоминающий файл, которым является наш объект ответа.

В результате в конце функции отображения мы получаем ответ с изображением в формате PNG, после чего мы просто устанавливаем соответствующий заголовок и отправляем ответ броузеру, который загрузит его.

На рис. 11.2 показано, как выглядит полученное изображение.

Разумеется, в действующем приложении наша функция извлекала бы данные из базы с помощью механизма ORM, а не пользовалась бы жестко определенными значениями. Для нужд своего приложения можно даже написать собственный тег шаблона, который извлекал бы данные из объекта QuerySet и отображал их в виде диаграммы в формате PNG. Кроме того, этот случай является прекрасным поводом для использования механизма кэширования, потому что операция создания графических файлов «на лету» потребляет немало вычислительных ресурсов.

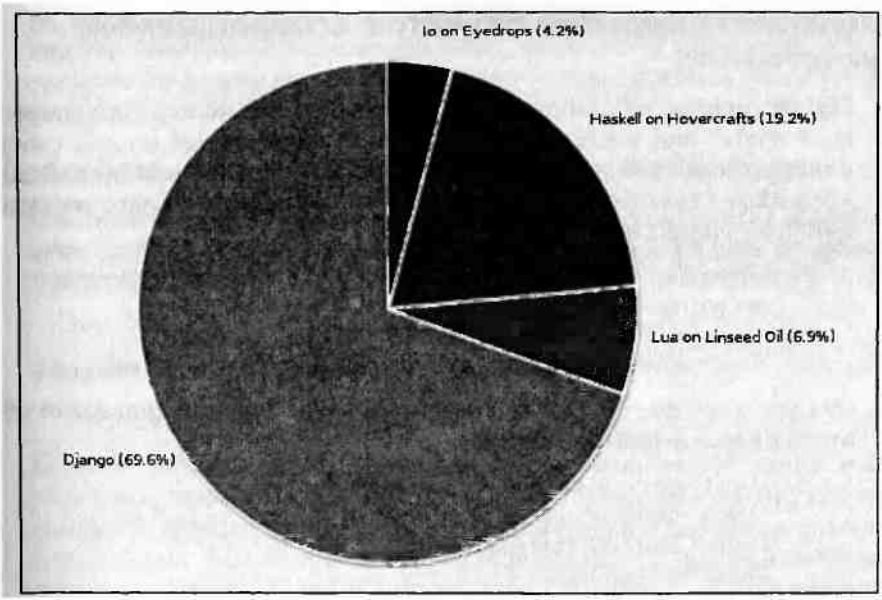

Рис. 11.2. Пример диаграммы, полученной с помощью библиотеки PyCha

Расширение механизма ORM с помощью собственных подклассов Manager

Хотя система ORM платформы Django не разрабатывалась для полной замены языка SQL, тем не менее, в большинстве случаев ее возможностей более чем достаточно для создания мощных веб-приложений. Вы уже знаете, как расширять возможности механизма ORM с помощью команд на языке SQL, передаваемых методу `extra`. Еще один способ предоставляют специализированные менеджеры. Вы уже использовали менеджеры, даже если и не знали этого. Рассмотрим пример:

```
really_good_posts = Post.objects.filter(gemlike=True)
```

Здесь `Post.objects` – это объект-менеджер. Это экземпляр класса, который наследует класс `models.Manager`, а методы этого класса определяют, что можно сделать с данными из базы данных – отфильтровать в данном случае.

Специализированный менеджер – это обычный класс, определяемый *вами*, который также наследует класс `models.Manager`. Он может быть полезен в двух случаях: когда необходимо изменить множество объектов, возвращаемых по умолчанию (обычно все объекты, хранящиеся в таблице), и когда требуется добавить новые методы манипулирования этим множеством объектов.

Изменение множества объектов, возвращаемых по умолчанию

Предположим, что, вместо того чтобы писать предыдущий запрос снова и снова, мы решили реализовать более краткий способ сообщить платформе Django, что вам необходимо извлечь из базы данных только «gemlike» (ценные) сообщения. Легко. Определим класс менеджера, как показано ниже:

```
class GemManager(models.Manager):
    def get_query_set(self):
        superclass = super(GemManager, self)
        return superclass.get_query_set().filter(gemlike=True)
```

Это все хорошо, но как его задействовать? Просто присвойте объект этого класса атрибуту модели.

```
class Post(models.Model):
    """Мой пример класса Post"""
    title = CharField(max_length=100)
    content = TextField()
    gemlike = BooleanField(default=False)

    objects = models.Manager()
    gems = GemManager()
```

Обратите внимание, что теперь модель имеет *два* менеджера – один с именем `objects` и другой с именем `gems`. Оба они возвращают объекты `QuerySet`, при этом менеджер `objects` ведет себя как менеджер по умолчанию с тем же именем.

Примечание

Наше определение менеджера `objects` эквивалентно определению, которое обычно автоматически создается платформой Django. Явное определение менеджера по умолчанию в этой модели продиктовано особенностями поведения Django при наличии дополнительных менеджеров – первый менеджер, который она обнаружит в модели, станет менеджером по умолчанию, который будет использоваться приложением администрирования для выборки объектов. Если опустить строку `objects = models.Manager()`, приложение администрирования не сможет отобразить сообщения, не являющиеся «ценными».

Познакомившись с реализацией нового менеджера `gems`, вероятно несложно догадаться, что он делает. С помощью функции `super` он вызывает метод `get_query_set` родительского класса (`models.Manager`) и затем фильтрует полученные результаты, как мы это уже делали прежде, до создания специализированного менеджера.

Как можно использовать его в программе? Как вы уже наверняка догадались, точно так же, как и менеджер по умолчанию `objects`.

```
really_good_posts = Post.gems.all()
```

И, конечно же, благодаря тому, что новый специализированный менеджер возвращает объект QuerySet, вы можете выполнять дополнительные операции над ним, применяя методы объекта QuerySet: filter, exclude и другие.

Добавление новых методов в подклассы Manager

Специализированный менеджер, который был определен выше, является удобным синтаксическим подсластителем. То есть он позволяет представить длинные запросы:

```
really_good_posts = Post.objects.filter(gemlike=True)
```

в более компактной форме:

```
really_good_posts = Post.gems()
```

Если бы нам приходилось выполнять массу операций с этим набором объектов, такой прием определенно помог бы повысить удобочитаемость программного кода, но и это еще не все о специализированных менеджерах. Мы можем добиться еще большего, добавив собственные методы в класс специализированного менеджера, принимающие дополнительные аргументы для обеспечения большей гибкости.

Продолжая наш искусственный пример, представим, что нам необходим компактный способ определить запрос, выбирающий только сообщения, содержащие определенное слово в заголовке и в тексте сообщения. Мы могли бы реализовать это с помощью привычного синтаксиса Django, как показано ниже:

```
cat_posts = Post.objects.filter(gemlike=True, title__contains="cats",
    content__contains="cats")
```

Запрос выглядит достаточно длинным, а если потребуется использовать его много раз, у вас быстро появится желание иметь какой-нибудь более компактный способ, например:

```
cat_posts = Post.objects.all_about("cats")
```

Примечание

Одна из интересных проблем, которые приходится решать при создании специализированных классов и методов, – это присваивание очевидных имен. Хорошее название метода, которое будет ясно читаться в цепочке Post.objects.foo, стоит того, чтобы затратить определенные усилия на его поиск – вам (и тем, кто будет пользоваться вашим программным кодом) часто придется видеть его. Как правило, в качестве имен менеджеров следует выбирать имена существительные («objects», «gems»), а в качестве имен методов – глаголы («exclude», «filter») или прилагательные («all», «latest», «blessed»).

Ниже приводится реализация метода менеджера, который использовался выше:

```
class AllAboutManager(models.Manager):
    """Возвращает только ценные сообщения, в которых идет речь об X"""
    def all_about(self, text):
        posts = super(AllAboutManager, self).get_query_set().filter(
            gemlike=True,
            title__contains=text,
            content__contains=text)
        return posts
```

Строго говоря, специализированные подклассы класса Manager не дают ничего такого, что нельзя было бы получить без их помощи. Однако чем очевиднее и понятнее прикладной интерфейс модели, тем проще сопровождать программный код, который использует его.

Расширение системы шаблонов

Шаблоны Django, с которыми вы познакомились в главе 6 «Шаблоны и обработка форм», реализуют два хорошо продуманных решения, которые не используются другими подобными системами. Как уже упоминалось, шаблоны могут воспроизводить содержимое в любом текстовом формате, а не только в разновидностях формата XML, таких как XHTML. Это делает шаблоны удобными для воспроизведения JavaScript, CSS, простого текста для сообщений электронной почты и других текстовых форматов.

Система шаблонов Django также отличается тем, что она не является воплощением или оберткой вокруг полноценного языка программирования. Это увеличивает скорость обработки шаблонов, снижает общую сложность платформы и обеспечивает простоту и ясность для дизайнеров страниц, не знакомых с программированием.

Для многих проектов встроенных возможностей системы шаблонов хватает с лихвой. Однако иногда возникают ситуации, когда необходимо или желательно иметь нечто большее. В этом разделе вы узнаете, как создавать специализированные теги шаблонов и фильтры и даже использовать сторонние системы шаблонов вместо той, что предоставляется платформой Django.

Простые специализированные теги шаблонов

Предположим, что нам необходимо вывести на главной странице сайта случайно выбранное изображение. Что, как вы же знаете, сделать достаточно легко. Для этого в функции представления создается список файлов изображений. Затем с помощью функции random.choice выбирается случайный файл и передается шаблону. Программный код такой функции представления может выглядеть примерно так:

```
def home_view(request):
    img_files = os.listdir(settings.RANDOM_IMG_DIR)
    img_name = random.choice(img_files)
```

```
img_src = os.path.join(settings.RANDOM_IMG_DIR, img_name)
# ... здесь выполняются прочие операции ...
render_to_response("home.html", {'img_src': img_src})
```

(Этот программный код следует в русле лучших традиций Django – хранить значения параметров настройки, такие как RANDOM_IMG_DIR, в файле settings.py, который может использоваться всеми приложениями проекта и изменяться по мере необходимости.)

Наконец, в вашем шаблоне должен иметься тег <img>, в котором используется значение, передаваемое в шаблон:

```
<img src="{{ img_src }}" />
```

Это все прекрасно. Но представим, что вы решили включить случайное изображение в *другую* страницу, которая создается *другой* функцией представления. Или ваш дизайнер заявил: «У меня есть еще пять страниц, где я мог бы вставлять случайные изображения... только в трех из них мне необходимо получить изображение из другого каталога... вы сможете это сделать?». Да, сможете!

Ключом к решению этой задачи является **специализированный тег шаблона**. Механизм, используемый платформой Django для создания своих тегов, также доступен и вам как программисту. То есть вы можете создать простой тег, которым сможет воспользоваться ваш дизайнер. Будет еще лучше, если тег сможет принимать путь к каталогу в качестве аргумента, – тогда дизайнер тоже сможет иметь свои «специальные» случаи.

Реализация тега выглядит чрезвычайно просто, поэтому мы сначала познакомим вас с программным кодом, а затем вернемся назад и разъясним детали. Требуемый нам тег выглядит примерно так:

```
<img src="{% random_image "faces/thumbnails" %}" />
```

Имя тега – random_image. Строка в кавычках – это путь к каталогу. Мы используем относительный путь, откладываемый от каталога MEDIA_ROOT. Система шаблонов платформы Django выполняет анализ аргументов и берет на себя заботу о передаче значений функции (поскольку теги шаблонов – это всего лишь функции).

Ниже приводится полный программный код реализации тега. Как всегда, нам требуется импортировать все модули, необходимые для решения поставленной задачи. Большая часть этого фрагмента – старый добрый Python – мы же будем пояснять только те особенности реализации, которые связаны с Django.

```
import os
import random
import posixpath
from django import template
from django.conf import settings

register = template.Library()
```

```
def files(path, types=[".jpg", ".jpeg", ".png", ".gif"]):
    fullpath = os.path.join(settings.MEDIA_ROOT, path)
    return [f for f in os.listdir(fullpath) if os.path.splitext(f)[1] in types]

@register.simple_tag
def random_image(path):
    pick = random.choice(files(path))
    return posixpath.join(settings.MEDIA_URL, path, pick)
```

Первой новинкой, на которую вы уже наверняка обратили внимание, является строка `register = template.Library()`. Экземпляр `template.Library()` обеспечивает нам доступ к декораторам, превращающим наши простые функции в теги и фильтры, которые могут использоваться системой шаблонов. Несмотря на то, что имя экземпляра класса может быть произвольным, то есть ему можно было бы присвоить любое другое имя и это никак не отразилось бы на работоспособности, тем не менее, в соответствии с соглашениями, принятыми в Django, настоятельно рекомендуется использовать имя `register`, так как это упростит понимание вашего программного кода другими.

Функция `files` – это обычная вспомогательная функция, возвращающая список имен файлов, расширения которых свидетельствуют о том, что они относятся к файлам изображений.

Функция `random_image` выполняется, когда наш тег используется в шаблоне, и принимает путь к каталогу, который указывается в теге. С помощью функции `files` она получает список имен файлов в указанном каталоге, выбирает одно из них, предваряет его значением параметра `MEDIA_URL`, чтобы получить путь к файлу, который может использоваться в теге `img`, и возвращает полный путь. (Функция `posixpath.join`, используемая здесь, не имеет никакого отношения к путям в стандарте POSIX – она просто объединяет составные части адреса URL, гарантируя, что между ними будет находиться единственный символ слеша, в отличие от `os.path.join`, которая использует начальные символы слеша даже в операционной системе Windows.)

Если в этом программном коде и есть что-то магическое, так это строка с декоратором `@register.simple_tag` перед функцией `random_image`. Он превращает нашу простую функцию в нечто, что может использоваться механизмом шаблонов.

Несмотря на то, что здесь мы определили всего один тег, в действительности созданный нами файл может представлять собой целую библиотеку тегов Django, в которой может содержаться множество тегов. Поэтому сохраните файл с именем, наиболее подходящим для библиотеки, например, `randomizers.py`, – в предположении, что мы будем добавлять в эту библиотеку другие теги, генерирующие случайное содержимое.

Файл следует сохранить в каталоге с именем `templatetags` где-нибудь в пути поиска, используемом системой шаблонов. То есть либо внутри одного из каталогов, перечисленных в параметре `INSTALLED_APPS` (если

в параметре settings.TEMPLATE_LOADERS присутствует значение django.template.loaders.app_directories.load_template_source), либо внутри одного из каталогов, перечисленных в параметре TEMPLATE_DIRS (если в параметре settings.TEMPLATE_LOADERS присутствует значение django.template.loaders.filesystem.load_template_source).

Ожидается, что каталог templatetags будет представлять собой пакет Python, что означает необходимость создания файла __init__.py внутри него – достаточно будет просто создать пустой файл с этим именем. Если вы забудете создать файл __init__.py в каталоге templatetags, вы получите сообщение об ошибке, которое, впрочем, однозначно идентифицирует проблему.

```
TemplateSyntaxError at /yourproject/
'randomizers' is not a valid tag library: Could not load template library
from django.templatetags.randomizers, No module named randomizers
(Перевод:
TemplateSyntaxError at /yourproject/
'randomizers' не является допустимой библиотекой тегов: Невозможно загрузить
библиотеку шаблона from django.templatetags.randomizers, Не найден модуль с
именем randomizers)
```

Если вы получите подобное сообщение об ошибке, создайте пустой файл __init__.py в каталоге templatetags. (Если вы забыли, зачем интерпретатору Python необходим этот файл, смотрите раздел, описывающий модули и пакеты в главе 1.)

Теперь ваш новый тег готов к использованию в любом приложении вашего проекта.

Чтобы сделать новый тег доступным в некотором шаблоне, добавьте в самое его начало тег load системы шаблонов. Тег load принимает один аргумент – имя модуля библиотеки (то есть имя файла без расширения «.py»).

```
{% load randomizers %}
```

Как только библиотека тегов будет загружена в шаблон, теги из этой библиотеки будут доступны шаблону, как если бы это были встроенные теги Django. Ваш новый тег random_image может принимать в виде аргумента либо литерал строки, либо переменную шаблона. Так, например, конкретный каталог, откуда следует выбирать случайные изображения, можно было бы определить внутри функции представления во время выполнения и передавать его шаблону в виде переменой image_dir. В этом случае тег random_image можно было бы использовать, как показано ниже:

```
<img src="{% random_image image_dir %}" />
```

Не менее просто можно определять теги, принимающие несколько аргументов. Декоратор simple_tag и система шаблонов Django сами выяснят, совпадает ли количество аргументов тега с ожидаемым.

Попробуем на основе предыдущего тега создать новый тег, выбирающий случайное изображение, который принимает второй аргумент. Допустим, что нам необходимо указывать точный тип (расширение) файла. Например, в нашем каталоге с изображениями могут храниться файлы в форматах PNG, GIF и JPEG и нам необходимо иметь возможность указывать, что выбирать следует только из файлов в формате PNG.

Ниже приводится новая функция, добавленная в файл randomizers.py.

```
@register.simple_tag
def random_file(path, ext):
    pick = random.choice(files(path, ["." + ext]))
    return posixpath.join(settings.MEDIA_URL, path, pick)
```

Новый тег называется random_file и использует вспомогательную функцию files из предыдущей версии. В этой новой версии функции просто был добавлен второй аргумент ext, который передается (в виде списка с одним элементом, с обязательным добавлением в начало символа «.») функции files в виде второго, необязательного аргумента.

Ниже демонстрируется порядок использования нашего нового тега:

```
<img src="{% random_file "special/icons" "png" %}" />
```

Для случаев, таких как этот, – когда желательно предоставить в распоряжение авторов шаблонов (включая и вас) компактный и удобочитаемый способ создания значений, которые в противном случае потребовали бы написания программного кода в нескольких функциях представлений, простые теги шаблонов могут оказаться просто незаменимыми. Если вы уже почувствовали желание познакомиться с чем-нибудь более сложным, читайте дальше.

Теги включения

В предыдущем примере теги возвращали простые строки. Если вам необходим специализированный тег, возвращающий более сложное информационное наполнение, – например, фрагмент разметки HTML, у вас может появиться желание создать функцию тега, создающую и возвращающую фрагмент разметки. Не делайте этого. Принципы архитектуры MVC (см. главу 3 «Начало») требуют хранить разметку HTML в шаблонах, а не в функциях представлений. Точно так же вы должны сохранять жестко определенную разметку HTML за пределами функций тегов шаблонов, если это возможно.

Для решения поставленной задачи можно было бы написать простой тег, используемый механизмом шаблонов, но платформа Django предоставляет более удобную и гибкую возможность: **теги включения**.

Теги включения являются наиболее удобными в ситуациях, когда необходимо отобразить фрагмент содержимого со значениями из текущего контекста шаблона. Например, допустим, что в шаблоне имеется

переменная {{ day }}, содержащая текущую дату, и вам необходим тег шаблона, который отображал бы простой календарь для текущего месяца.

Попробуем создать такой тег прямо сейчас. Наш тег будет базироваться на модуле calendar языка Python, который способен представлять месяц в виде списка, содержащего номера дней (дни, принадлежащие соседним месяцам, отмечаются числом 0).

```
>>> import calendar
>>> calendar.monthcalendar(2010, 7)
[[0, 0, 0, 1, 2, 3, 4], [5, 6, 7, 8, 9, 10, 11], [12, 13, 14, 15...
```

Теперь нам необходимо некоторое преобразование, которое превратит нулевые значения в полученном списке в пустые строки, потому что нам не нужно отображать дни, предшествующие началу месяца и следующие за концом месяца, в виде нулей. Для обработки списка, возвращаемого функцией calendar.monthcalendar, мы используем генератор списков.

```
>>> import calendar
>>> month = calendar.monthcalendar(2010, 7)
>>> [[day or '' for day in week] for week in month]
[['', '', '', 1, 2, 3, 4], [5, 6, 7, 8, 9, 10, 11], [12, 13, 14...
```

Этот довольно мудреный генератор списков выполняет обход всех недель в месяце, для каждой недели выполняет обход всех дней и, если встречается ненулевое значение (значение True в логическом контексте), возвращает *это* значение, в противном случае – пустую строку.

Так как механизм шаблонов никак не заботится о том, передаются ли целые числа или строки, такое смешение типов данных не будет вызывать ошибок. Если же мы передаем эти данные функции на языке Python для дальнейшей обработки, такое смешение может вызвать некоторые проблемы.

Модуль calendar может возвращать даже названия дней недели и месяцев.

```
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July'...
```

На данном этапе развития HTML календари по-прежнему представляют собой табличные данные, поэтому для представления календаря в разметке HTML мы будем использовать тег <table>. Создадим небольшой шаблон исключительно для нашего календаря, предполагая, что полный список передается в виде переменной weeks:

```
<table>
    <tr><th colspan='7'>{{ month }} {{ year }}</th></tr>
    <tr>{% for dayname in daynames %}<th>{{ dayname }}</th>{% endfor %}</tr>
```

```
    {% for week in weeks %}
        <tr>
        {% for day in week %}
            <td>{{ day }}</td>
        {% endfor %}
        </tr>
    {% endfor %}
</table>
```

Мы предпочитаем следовать соглашениям об именовании частичных шаблонов, включая шаблоны, подобные этому, – которые будут использоваться в тегах включения, и начинать их имена с символа подчеркивания. Это служит визуальным напоминанием, что такие шаблоны не предназначены, чтобы служить основой полноценного документа. Назовите свой фрагмент шаблона `_calendar.html` и сохраните его где-нибудь, где его сможет отыскать система шаблонов, – например, в каталоге `templates` приложения или в одном из каталогов, перечисленных в параметре настройки `TEMPLATE_DIRS`.

Теперь перейдем к функции тега включения. Создайте в каталоге `templatetags` новый файл с именем `inclusion_tags.py`.

```
@register.inclusion_tag("_calendar.html")
def calendar_table():
    import calendar
    import datetime
    date = datetime.date.today()
    month = calendar.monthcalendar(date.year, date.month)
    weeks = [[day or '' for day in week] for week in month]
    return {
        'month': calendar.month_name[date.month],
        'year': date.year,
        'weeks': weeks,
        'daynames': calendar.day_abbr,
    }
```

Обратите внимание, что наша функция возвращает словарь – он играет роль словаря контекста при отображении шаблона, передаваемого функции `@register.inclusion_tag`. Другими словами, любой ключ этого словаря превращается в переменную шаблона, которая может использоваться для отображения соответствующего значения.

Кроме уже сказанного, о тегах включения практически нечего добавить. Они предоставляют удобный способ, позволяющий отделять представление от логики. Вы могли бы создать простой тег с помощью декоратора `simple_tag`, который создает и возвращает большой фрагмент разметки HTML, но такое решение было бы сложно сопровождать. Вы могли бы создать тег, который сам производит обращения к механизму шаблонов, но для этого пришлось бы писать шаблонный программный код. Теги включения позволяют создавать более компактные реализации и сохранять содержимое шаблонов в виде шаблонов.

Мы добавили в нашу страницу таблицу стилей, созданную на скорую руку, чтобы немножко приукрасить наш календарь. Добавьте следующий код разметки в начало файла _calendar.html.

```
<style type="text/css">
td, th { padding: 4px; width: 30px; background: #bbb; }
td { text-align: right; }
</style>
```

Когда вам потребуется задействовать этот тег, просто добавьте строку {% load inclusion_tags %} в начало шаблона, где используется этот тег. Чтобы добавить календарь в страницу, в нужном месте шаблона просто добавьте тег {% calendar_table %}.

Таблица календаря, в зависимости от броузера, выглядит, как показано на рис. 11.3.

Рис. 11.3. Наш календарь

Неплохо! Благодаря удачному отделению управляющей логики от представления ваш дизайнер легко сможет подкорректировать файл шаблона тега. А в это время вы или другие разработчики сможете вносить изменения в содержимое – например, выполнить локализацию названий месяцев и дней недели, вообще не прикасаясь к шаблону (или к шаблонам, включающим его).

Специализированные фильтры

В состав системы шаблонов платформы Django входит большое число полезных фильтров, но когда-нибудь у вас может появиться причина добавить свой собственный фильтр. Фильтры просты в использовании, и их так же просто создавать. Как и в случае тегов, для определения фильтров можно использовать удобный синтаксис декораторов.

Фильтры – это обычные функции, которые в большинстве случаев принимают и возвращают строки. Интересным и не очень сложным примером (он может показаться сложным, если вы неуверенно чувствуете себя при работе с регулярными выражениями) может служит

фильтр wikify, преобразующий слова с ПеременнымРегистромСимволов в ссылки HTML, пригодные для использования в wiki. Ниже приводится полная реализация фильтра:

```
import re
from django.template import Library
from django.conf import settings

register = Library()
wikifier = re.compile(r'\b(([A-Z]+[a-z]+){2,})\b')

@register.filter
def wikify(text):
    return wikifier.sub(r'<a href="/wiki/\1/">\1</a>', text)
```

Этот фильтр ожидает получить строку, в которой все вхождения слов с ПеременнымРегистромСимволов замещаются ссылками на /wiki/ПеременнымРегистромСимволов. Его можно было бы использовать, например, в следующем шаблоне (предполагается, что переменные title и content хранят заголовок страницы и содержимое с разметкой wiki соответственно).

```
{% load wikitags %}
<h1>My Amazing Wiki Page: {{ title }}</h1>
<div class="wikicontent">
{{ content|wikify }}
</div>
```

Фильтры и дополнительные аргументы

Предыдущая функция wikify принимает единственный аргумент, значением которого может быть значение любого выражения, стоящего слева от символа «|», предшествующего имени фильтра. Но как быть, если необходимо иметь возможность корректировать действие фильтра с помощью дополнительного аргумента?

Фильтры можно писать так, что они будут принимать второй аргумент. Часто такой второй аргумент используется для корректировки действия фильтра. Например, предположим, что вам необходима функция, которая обеспечивала бы отображение строки только в случае, если эта строка содержит определенную последовательность символов (вспомните команду grep в системе UNIX). Конечно, то же самое можно было бы реализовать с помощью тегов шаблона if/then, но фильтр мог бы обеспечить более компактную форму записи.

```
{{ my_string|grep:"magic" }}
```

Определение такого фильтра выглядит, как показано ниже:

```
@register.filter
def grep(text, term):
    if text.contains(term):
        return text
```

Аргументы фильтров всегда заключаются в кавычки и отделяются от имени фильтра двоеточием. Даже если вы используете числовое или другое значение, не являющееся строкой, синтаксис шаблонов Django требует заключать их в кавычки. То есть, если вы создали фильтр, который обеспечивает вывод входной строки, только если ее длина не превышает определенного числа, его использование выглядит примерно так, как показано ниже:

```
{{ bla_bla_bla|hide_if_shorter_than:"100" }}
```

При реализации фильтра, такого как этот, следует учитывать необходимость преобразования типов. Так, для преобразования значения второго аргумента мы использовали функцию `int`.

```
@register.filter
def hide_if_shorter_than(text, min_len):
    if len(text) >= int(min_len):
        return text
```

Здесь мы выполняем явное преобразование аргумента `min_len` в тип `int`, потому что значением этого аргумента является строка, – по причинам, описанным выше.

Источником заблуждений, вероятно, также может быть первый аргумент, передаваемый в функцию фильтра, – это фактическое значение, которое требуется «профильтровать», или модифицировать. Оно необязательно должно быть строкой. Это, например, справедливо с точки зрения фильтров, связанных с преобразованием даты и времени, оперирующих объектами `datetime`. Это также означает, что имеется возможность создавать фильтры, оперирующие объектами, отличными от строк. Это не совсем обычное явление – просто потому, что веб-разработка тесно связана с манипулированием строками, но это возможно.

Если вы знаете, что ваша функция фильтра может принимать какие-либо не строковые данные, но вам *требуется* интерпретировать их как строки, вы можете добавить к своей функции декоратор `stringvalue`. К одной и той же функции допускается применять несколько декораторов, поэтому, если мы решили бы добавить этот декоратор к нашей функции `hide_if_shorter_than`, мы просто добавили бы декоратор `@stringvalue` в строке под декоратором `@register.filter`.

Примечание

Порядок применения декораторов к функции имеет значение. Порядок следования, упоминавшийся выше, когда декоратор `@register.filter` предшествует декоратору `@stringvalue`, за которым следует сама функция, обеспечивает передачу строки в фильтр. Обратный порядок, когда первым применяется декоратор `@stringvalue`, а за ним следует декоратор `@register.filter`, гарантирует лишь, что *возвращаемое значение* фильтра будет строкой. Тонкое, но важное отличие.

Более сложные специализированные теги шаблонов

Существует возможность создавать более сложные теги шаблонов, например пары блочных тегов, выполняющих некоторые преобразования содержимого, заключенного между ними. Создание таких тегов – задача не из легких, связанная с выполнением непосредственных манипуляций с внутренними механизмами системы шаблонов Django. Разработка таких тегов более трудоемка, чем простых тегов, и требуется не очень часто. За дополнительной информацией о создании своих собственных тегов шаблонов обращайтесь к электронной документации к платформе Django по адресу *www.djangoproject.com/documentation/templates_python*.

Альтернативные системы шаблонов

Работа механизма шаблонов Django состоит в том, чтобы подготовить строки, которые будут играть роль содержимого объектов `HttpResponse`. Как только вы начинаете это понимать, становится очевидным, что можно использовать и другой механизм шаблонов, если по каким-либо причинам вы пришли к выводу, что не в состоянии достигнуть поставленных целей с помощью языка шаблонов Django.

Простой текст

Ниже приводится простейший из возможных альтернативных механизмов шаблонов для Django.

```
def simple_template_view(request, name):
    template = "Hello, your IP address is %s."
    return HttpResponse(template % request.META['REMOTE_ADDR'])
```

Не требуется никаких сторонних модулей – эта функция представления просто использует синтаксис строковых шаблонов языка Python.

Выбор альтернативного механизма шаблонов

Сила платформы Django в том, что она реализует полный пакет интегрированных механизмов. Однако она не является неким монолитным образованием – если вам потребуется заменить компонент Django сторонним пакетом, который вам больше подходит, в большинстве случаев это вполне реализуемо.

Система шаблонов может быть заменена особенно легко. Какие причины могли бы подтолкнуть к использованию других систем шаблонов?

- Вы перешли на платформу Django, имея опыт работы с другой системой, синтаксис которой кажется вам более удобным.
- Вы занимаетесь разработкой других проектов на других веб-платформах, написанных на языке Python, и вам хотелось бы использовать единый язык шаблонов.

- Вы переводите проект с другой веб-платформы, написанной на языке Python, на платформу Django, но у вас не хватает времени на преобразование существующих шаблонов.
- Ваша логика отображения требует некоторых особенностей, добавить которые в язык шаблонов Django не так-то просто.

Использование других механизмов шаблонов: Mako

Одним из популярных механизмов шаблонов, который мы будем использовать в этом разделе, является Mako. Он существенно отличается от языка шаблонов Django как по внешнему виду, так и по своей архитектуре, но обладает похожими достоинствами: высокая скорость обработки шаблона, не ограничен рамками, присущими XML-подобным языкам, и имеет похожий механизм наследования.

Механизм Mako пришел на смену платформе шаблонов, написанной на языке Python, под названием Myghty, которая в свою очередь базировалась на известной системе `HTML::Mason`, написанной на языке Perl. Механизм Mako используется такими сайтами, как *reddit.com* и *python.org*, и в настоящее время является механизмом шаблонов по умолчанию в другой веб-платформе, написанной на языке Python, – Pylons. Поэтому, если вы ищете альтернативную систему шаблонов, Mako является отличным кандидатом на рассмотрение. Кроме того, этот механизм оказал существенное влияние на систему шаблонов Django, поэтому, несмотря на отличия в синтаксисе, между этими двумя системами существует определенное концептуальное сходство, которое делает миграцию между ними проще, чем могло бы показаться.

В отличие от шаблонов Django, синтаксис Mako основан на языке Python. Это существенное отклонение от философии шаблонов, проповедуемой платформой Django, которая стремится ограничить присутствие программной логики в шаблонах. Оба подхода имеют свои положительные стороны. Идея, лежащая в основе Mako, состоит в том, чтобы сделать ее проще для программистов, использующих язык Python; при этом ясность синтаксиса Python делает его доступным для дизайнеров шаблонов.

Прежде чем мы создадим функцию представления Django, использующую механизм шаблонов Mako, рассмотрим простой пример в интерактивной оболочке интерпретатора, который позволит вам почувствовать, как работает механизм Mako.

```
>>> from mako.template import Template
>>> t = Template("My favorite ice cream is ${name}")
>>> t.render(name="Herrell's")
"My favorite ice cream is Herrell's"
>>> context = {'name': "Herrell's"}
>>> t.render(**context)
"My favorite ice cream is Herrell's"
```

В первом примере мы явно передаем имя фактической переменной, а во втором используем контекст, как это делали ранее. Не забывайте, что контекст – это просто словарь, который передается методу render. Такой подход должен показаться вам знакомым – он практически идентичен способу, используемому механизмом шаблонов Django.

В Mako также имеются фильтры, по своему синтаксису сильно напоминающие фильтры Django.

```
>>> from mako.template import Template
>>> t = Template("My favorite ice cream is ${name | entity}")
>>> t.render(name="Emack & Bolio's")
"My favorite ice cream is Emack & Bolio's"
```

Теперь можно задействовать шаблон Mako в функции представления Django.

```
from mako.template import Template

def mako_view(request):
    t = Template("Your IP address is ${REMOTE_ADDR}")
    output = t.render(**request.META)
    return HttpResponse(output)
```

В этом представлении мы делаем то же самое, что и в предыдущем интерактивном примере, – создаем новый шаблон Mako и отображаем его, используя в качестве контекста объект META, полученный в запросе, с которым мы коротко познакомились в главе 5 «Адреса URL, механизмы HTTP и представления».

Если вы действительно собираетесь использовать механизм Mako, сохраняйте свои шаблоны в файловой системе (или в базе данных) так, чтобы платформа Django смогла отыскивать их, как она отыскивает свои собственные шаблоны (без необходимости указывать полные пути), создайте метод render_to_response, дружественный по отношению к Mako, и т. д. К счастью, большая часть этой работы уже выполнена до вас другими исследователями Mako/Django. На сайте Django Snippets (*http://www.djangosnippets.org/snippets/97/*) вы найдете интересные фрагменты программного кода и сопровождающие их пояснения, которые можете скопировать и опробовать.

В заключение

Как упоминалось во введении, эта глава рассказывает обо всем понемногу, но мы надеемся, что сумели открыть некоторые двери для вас как для разработчика веб-приложений на платформе Django и наглядно показать, насколько гибкой и расширяемой в действительности является платформа Django. Следующая глава завершает четвертую часть книги еще одной подборкой разделов на передовые темы.

12

Передовые приемы развертывания Django

Как и глава 11 «Передовые приемы программирования в Django», эта глава состоит из нескольких не связанных между собою разделов по разным темам. В главе 11 рассматривались темы, касающиеся разработки программного кода. Здесь будут рассматриваться темы, не имеющие прямого отношения к программированию и связанные с развертыванием ваших приложений, изменением среды окружения, в которой они работают, и с модификацией самой платформы Django.

Создание вспомогательных сценариев

Django – это веб-платформа, но это не означает, что у вас отсутствует возможность взаимодействовать с ней без броузера. В действительности, одно из важнейших преимуществ платформы Django, написанной на языке Python, а не на одном из языков веб-программирования, таких как ColdFusion или PHP, состоит в том, что она предусматривает возможность взаимодействия с ней из командной строки. Нередко может возникать потребность периодически выполнять некоторые операции над данными, которыми управляет ваше приложение на платформе Django, не создавая полный веб-интерфейс.

Ниже приводятся некоторые типичные случаи, когда предпочтительнее пользоваться вспомогательными сценариями, опирающимися на Django:

- Создание кэша значений или документов, который затем перестраивается каждую ночь (или каждый час)
- Импортирование данных в модели Django
- Отправка запланированных уведомлений по электронной почте

- Создание отчетов
- Выполнение операций по очистке устаревших данных (например, удаление зависших сеансов)

Это та самая сторона использования Django, когда твердые навыки владения языком Python имеют особую ценность. При создании вспомогательных сценариев вы просто используете язык программирования Python и выполняете некоторые настройки среды окружения, необходимой для платформы Django.

Ниже приводятся несколько примеров таких вспомогательных сценариев. Вслед за программным кодом каждого из них следует описание; это поможет вам определить, какой из подходов больше подходит для вашего проекта.

Задания cron, выполняющие очистку

В интенсивно используемых базах данных SQLite (и некоторых PostgreSQL), когда часто удаляются старые записи и создаются новые, полезно производить периодическую «очистку», чтобы освободить неиспользуемое пространство. Например, на сайте *dpaste.com* большая часть записей сохраняется в базе данных в течение месяца, после чего они удаляются. Это означает, что каждую неделю обновляется примерно 25 процентов записей.

Без периодической очистки база данных выросла бы до гигантских размеров. Хотя, по утверждениям разработчиков, база данных SQLite способна поддерживать объем файлов базы данных до 4 Гбайт, лучше было бы не проверять ее работоспособность на пределе этого размера. Ниже показано, как выглядит сценарий, выполняющий очистку на сайте *dpaste.com*. Он запускается каждую ночь с помощью планировщика `cron`. (В операционной системе Windows его автоматически можно запускать как «службу».)

```
import os
import sys
os.environ['DJANGO_SETTINGS_MODULE'] = "dpaste.settings"
from django.conf import settings

def vacuum_db():
    from django.db import connection
    cursor = connection.cursor()
    cursor.execute("VACUUM")
    connection.close()

if __name__ == "__main__":
    print "Vacuuming database... "
    before = os.stat(settings.DATABASE_NAME).st_size
    print "Size before: %s bytes" % before
    vacuum_db()
    after = os.stat(settings.DATABASE_NAME).st_size
```

```
print "Size after: %s bytes" % after
print "Reclaimed: %s bytes" % (before - after)
```

В самом начале этого сценария после первых двух инструкций импортирования выполняется настройка среды окружения, в частности, устанавливается значение наиболее важной переменной окружения DJANGO_SETTINGS_MODULE, чтобы указать платформе Django, с каким проектом мы будем работать.

Этот сценарий предполагает, что сама платформа Django и ваш проект находятся в пути поиска, который используется интерпретатором Python при поиске модулей. Они могут быть связаны символическими ссылками из каталога site-packages, установлены как пакеты Python или включены в переменную окружения PYTHONPATH. Если вам необходимо указать пути вручную, добавьте следующие строки сразу вслед за первыми двумя инструкциями импортирования:

```
sys.path.append('/YOUR/DJANGO/CODEBASE')
sys.path.append('/YOUR/DJANGO/PROJECTS')
```

Конечно, не забудьте указать свои пути к каталогам – первый указывает местоположение исходных текстов Django в системе (как и все исследователи Django, мы используем самую последнюю версию Django, полученную из репозитория Subversion), а второй добавляет в переменную sys.path каталог проектов, благодаря чему все наши проекты обнаруживаются инструкциями import, ссылающимися на них.

При создании вспомогательных сценариев, опирающихся на использование платформы Django, важно помнить, что в конечном счете – это всего лишь сценарии на языке Python. Пока интерпретатор Python знает, где находится ваш файл с настройками, вы можете считать себя полностью подготовленным.

Импорт/экспорт данных

Командная строка отлично подходит для создания инструментов, используемых редко и недоступных конечным пользователям. Например, если периодически вы получаете некоторые данные, которые требуется добавлять в базу данных, можно написать вспомогательный сценарий, выполняющий эту работу.

Если ваш проект основан на использовании базы данных SQL, у вас может появиться вопрос, зачем идти кружным путем создания сценария Python/Django для импорта данных, если вместо этого можно просто использовать язык SQL.

Дело в том, что, как правило, поступающие данные требуют некоторой предварительной обработки, прежде чем они смогут быть преобразованы в код SQL. Фактически, если вы предполагаете импортировать данные в каком-либо формате более чем пару раз, будет проще написать инструмент, способный напрямую воспринимать данные в представ-

ленном формате (CSV, XML, JSON, простой текст или какой-либо иной формат), чем выполнять набор операций поиска с заменой в текстовом редакторе, пытаясь встроить требуемые данные в инструкции INSERT языка SQL.

Подобные задачи входят в круг тех, где эффективно срабатывает принцип Python: «батарейки входят в комплект» – в состав стандартной библиотеки языка Python входят модули, предназначенные для работы с весьма широким разнообразием форматов файлов. Например, если необходимо создать архив электронной почты и требуется импортировать файлы «mbox», характерные для системы UNIX, можно воспользоваться модулем email, входящим в стандартную библиотеку, – вместо того, чтобы писать свой уникальный, но требующий времени и усилий на написание или на отлаживание (или и на то, и на другое) парсер.

Ниже приводится простая модель, которая может использоваться для сохранения электронных писем. Она весьма схожа с моделью, используемой на сайте *purportal.com* для архива, где хранится «жульнический спам».

```
class Message(models.Model):
    subject = models.CharField(max_length=250)
    date = models.DateField()
    body = models.TextField()
```

Предположим, что у вас имеется этот модуль и файл почтового ящика mbox, путь к которому указан в параметре settings.MAILBOX настройки проекта; тогда можно было бы импортировать электронные письма в модель, как показано ниже:

```
import os, mailbox, email, datetime
try:
    from email.utils import parsedate # Python >= 2.5
except ImportError:
    from email.Utils import parsedate # Python < 2.5

os.environ['DJANGO_SETTINGS_MODULE'] = "YOURPROJECT.settings"
from django.conf import settings
from YOURAPP.models import Message

mbox = open(settings.MAILBOX, 'rb')
for message in mailbox.PortableUnixMailbox(mbox, email.message_from_file):
    date = datetime.datetime(*parsedate(message['date'])[:6])
    msg = Message(
        subject=message['subject'],
        date=date,
        body=message.get_payload(decode=False),
    )
    msg.save()

print "Archive now contains %s messages" % Message.objects.count()
# При необходимости теперь можно очистить файл ящика mbox:
# open(MAILBOX, "w").write("")
```

Как уже упоминалось, это лишь небольшой пример, как писать сценарии в рамках проектов на платформе Django. Стандартная библиотека языка Python, не говоря уже об имеющейся коллекции сторонних библиотек, позволяет решать чрезвычайно широкий круг задач. Если вы планируете всерьез заниматься разработкой приложений на платформе Django, вам определенно стоит потратить время и поближе познакомиться со стандартной библиотекой Python (*http://docs.python.org/lib/*), чтобы иметь представление о том, какие возможности она предоставляет.

Изменение программного кода самой платформы Django

Изменение программного кода самой платформы Django является крайней мерой. Не потому, что это сложно, – в конце концов, это всего лишь программный код на языке Python, к тому же содержащий большой объем встроенной документации в виде строк документирования и комментариев. Причина, по которой мы не рекомендуем вам вторгаться во внутреннее устройство Django, чтобы «исправить» некоторую неувязку, состоит в том, что зачастую это не стоит затраченных усилий.

Django – это проект, который продолжает активно развиваться. Поскольку во главу угла поставлена стабильность, главная версия Django обладает высокой надежностью. По мере добавления новых особенностей и исправления старых ошибок на сайте *code.djangoproject.com* появляются соответствующие сообщения, ознакомившись с которыми вы можете обновить свою версию платформы, когда пожелаете. Однако, если вы внесете свои исправления непосредственно в программный код платформы, вы тем самым закроете для себя возможность выполнять обновления. Или, в лучшем случае, вам придется потратить немало усилий на объединение новых обновлений со своими изменениями. Хотя, если это действительно необходимо, такую работу могут облегчить распределенные системы управления версиями. (Подробнее об этом подходе рассказывается в приложении С «Инструменты разработки для платформы Django».)

Наконец, если вы чувствуете неодолимое стремление внести исправления в программный код Django, подумайте о том, будут ли полезны для других пользователей изменения, которые вы внесли, преследуя собственные цели. Если на ваш взгляд это действительно так, то прочитайте раздел «Помощь проекту Django» в приложении Е «Участие в проекте Django».

Кэширование

Производительность сайтов, испытывающих высокие нагрузки, редко ограничивается недостаточной возможностью веб-сервера быстро от-

правлять данные. Практически всегда узким местом является воспроизведение этих данных – база данных не способна отвечать на запросы достаточно быстро или центральный процессор может быть перегружен постоянным выполнением одного и того же программного кода для каждого запроса. Решение этой проблемы заключается в организации **кэширования** – сохранении копии сгенерированных данных, благодаря чему устраняется необходимость каждый раз выполнять «дорогостоящие» запросы к базе данных или вычисления.

Для сайтов с высокой нагрузкой кэширование – это насущная необходимость независимо от того, какая технология положена в основу. Платформа Django имеет обширную поддержку механизмов кэширования с тремя уровнями управления, которые могут задействоваться в зависимости от потребностей вашего сайта. Кроме того, имеется удобный тег шаблонов, позволяющий указывать, какие разделы отображаемых страниц должны кэшироваться.

Типичный пример кэширования

Средства кэширования платформы Django предоставляют подавляющее воображение начинающего пользователя огромное число возможных конфигураций. Хотя потребности каждого сайта (как и возможности каждого сервера) различны, вы лучше поймете, как использовать этот инструмент, если мы начнем с конкретного примера. В качестве бонуса вы получите конфигурацию, которая с успехом может использоваться большим количеством сайтов, поэтому, возможно, вам больше ничего и не понадобится знать об организации кэширования в Django.

Точка отсчета

Суть кэширования состоит в том, чтобы увеличить производительность сайта, поэтому имеет смысл выполнить некоторые предварительные измерения производительности. Все сайты отличаются друг от друга, и единственный способ узнать эффект применения кэширования к *вашему* сайту – это оценить его.

Одним из основных инструментов проведения базового тестирования производительности сервера является `ab` – инструмент тестирования производительности проекта Apache. Если вы используете веб-сервер Apache, следовательно, инструмент `ab` у вас уже имеется. В любой POSIX-совместимой системе, такой как Linux или Mac OS X, он должен находиться в одном из каталогов, включенных в переменную окружения `PATH`. В системе Windows его можно найти в каталоге установки Apache, например `C:\Program Files\Apache Software Foundation\Apache2.2\bin`. (За дополнительной информацией о порядке использования обращайтесь к странице руководства по адресу *http://httpd.apache.org/docs/2.2/programs/ab.html*.)

Работает этот инструмент следующим образом: вы передаете ему адрес URL и число запросов, которые следует произвести, а он возвращает вам информацию о производительности. Ниже приводятся результаты тестирования инструментом ab нашего примера приложения блога из главы 2 «Django для нетерпеливых: создание блога». В последней строке выводится число «запросов в секунду». Пусть вас не смущают конкретные числа, которые приводятся в этом примере, потому что измерения производились на ноутбуке, приобретенном еще три года тому назад, – мы полагаем, что ваш сервер обладает куда более высокой производительностью!

```
$ ab -n 1000 http://127.0.0.1:8000/blog/
...
Benchmarking 127.0.0.1 (be patient)
...
Finished 1000 requests
...
Time taken for tests: 27.724 seconds
...
Requests per second: 36.07 [#/sec] (mean)
(Перевод:
Тестируется 127.0.0.1 (наберитесь терпения)
...
Выполнено 1000 запросов
...
Время тестирования: 27.724 секунд
...
Запросов в секунду: 36.07 [#/sec] (в среднем))
```

Итак, мы получили порядка 36 запросов в секунду. Теперь активируем механизм кэширования и посмотрим, что изменилось.

Добавляем промежуточную обработку

Средства кэширования в Django являются частью механизма промежуточной обработки и неактивны по умолчанию. Чтобы задействовать их, необходимо открыть файл settings.py и добавить в параметр MIDDLEWARE_CLASSES строку django.middleware.cache.CacheMiddleware. Вообще говоря, ее нужно добавить в самый конец, потому что некоторые другие средства промежуточной обработки (особенно это относится к SessionMiddleware и GZipMiddleware) могут конфликтовать с заголовком HTTP Vary, на использование которого опирается механизм кэширования.

Установка типа кэширования

Платформа кэширования предлагает как минимум четыре механизма хранения данных. Для простоты мы будем использовать заданный по умолчанию внутренний кэш Django – кэш в локальной памяти с именем locmem. При использовании этого механизма кэшируемые данные сохраняются в оперативной памяти, которая обеспечивает чрезвычай-

но высокую скорость поиска. По сравнению со многими другими реализациями кэширования, сохраняющими кэш на диске, кэш в оперативной памяти может обеспечить весьма существенный прирост производительности. (Если вы скептически относитесь к этим заявлениям, смотрите ниже обсуждение Memcached, чрезвычайно высокопроизводительного механизма кэширования, который первоначально создавался для поддержки сайта *LiveJournal.com*.)

Добавьте в свой файл settings.py следующую строку:

```
CACHE_BACKEND = "locmem://"
```

(Непривычный вид псевдо-URL в этом параметре приобретет для вас больше смысла, когда вы познакомитесь с другими механизмами, использующими формат URL для передачи аргументов конфигурации. Это значение является значением по умолчанию, поэтому, строго говоря, мы можем не устанавливать этот параметр, если нам не требуется определить какой-то другой механизм кэширования. Однако один из основных принципов Python гласит: «Явное лучше неявного», а кроме того, перейти на другой механизм кэширования с другими параметрами настройки будет проще, если конфигурационный параметр уже присутствует в файле.)

Опробование

Это все, что необходимо в простейшем случае реализации кэширования в пределах всего сайта. Теперь давайте проверим, как изменилась производительность сайта с активированным механизмом кэширования.

```
$ ab -n 1000 http://127.0.0.1:8000/blog/
...
Benchmarking 127.0.0.1 (be patient)
...
Finished 1000 requests
...
Time taken for tests: 8.750 seconds
...
Requests per second: 114.29 [#/sec] (mean)
(Перевод:
Тестируется 127.0.0.1 (наберитесь терпения)
...
Выполнено 1000 запросов
...
Время тестирования: 8.750 секунд
...
Запросов в секунду: 114.29 [#/sec] (в среднем))
```

Производительность увеличилась более чем *в три раза*, и для этого нам потребовалось добавить всего две строки в файл settings.py. Однако, имейте в виду, что наше приложение блога не выполняет тяжелых запросов к базе данных и не реализует сложную бизнес-логику. На бо-

лее сложных приложениях можно ожидать более существенный прирост производительности.

Стратегии кэширования

Хотя предыдущая, самая простая реализация кэширования позволяет получить впечатляющие результаты, тем не менее, она подходит не для всех ситуаций. Мы не рассматривали, как *долго* хранятся данные в кэше, как кэшировать содержимое, которое не является полной веб-страницей (например, сложные врезки и виджеты), как избежать кэширования отдельных страниц (например, страниц административного раздела) или какие аргументы можно использовать для управления производительностью. Давайте поговорим сейчас о некоторых из этих ситуаций.

Кэширование всего сайта

Механизм, активированный нами выше, называется механизмом кэширования всего сайта. Платформа Django просто кэширует результаты всех запросов, в которых отсутствуют аргументы запросов GET и POST. Мы рассмотрели самый простой вариант использования механизма кэширования, но в файле `settings.py` существует еще несколько других параметров, которые помогут вам выполнить его настройку.

- `CACHE_MIDDLEWARE_SECONDS`: Интервал времени в секундах, в течение которого страница будет храниться в кэше, прежде чем будет замещена более свежей копией. Значание по умолчанию составляет 600 секунд (десять минут).
- `CACHE_MIDDLEWARE_KEY_PREFIX`: Строка, которая будет использоваться в качестве префикса к элементам в кэше. Когда используется один кэш для нескольких сайтов – в памяти, в файлах или в базе данных, этот ключ гарантирует отсутствие конфликтов между сайтами. Вы можете использовать в этом параметре любую уникальную строку – доменное имя сайта или `str(settings.SITE_ID)` – любой из этих вариантов будет разумным выбором.
- `CACHE_MIDDLEWARE_ANONYMOUS_ONLY`: Простое кэширование, основанное на адресах URL, не всегда дает желаемые результаты при использовании в интерактивных веб-приложениях, где содержимое страниц с одним и тем же адресом URL часто изменяется в ответ на действия пользователя. Даже если общедоступный раздел вашего сайта не содержит информационного наполнения, добавляемого пользователем, при использовании приложения администрирования вам может потребоваться установить этот параметр в значение `True`, чтобы гарантировать, что ваши изменения (добавление, удаление или изменение содержимого) будут немедленно отражаться на страницах административного раздела.

Если работа механизма кэширования удовлетворяет вашим потребностям, то описанного выше вам будет вполне достаточно. Однако такой

вариант кэширования подходит не для всех случаев. Давайте посмотрим, какие еще параметры кэширования предлагаются платформой Django, и в каких случаях их преимуществами можно воспользоваться.

Кэширование отдельных представлений

Механизм кэширования всего сайта предполагает, что все его части будут храниться в кэше в течение одного и того же интервала времени. Однако у вас могут быть другие требования. Например, предположим, что вы сопровождаете новостной сайт и следите за популярностью отдельных статей, собирая статистику посещений с целью создания списков наиболее популярных страниц. Совершенно очевидно, что список «Лучшие статьи за вчерашний день» может храниться в кэше 24 часа. С другой стороны, список «Лучшие статьи за сегодня» может изменяться в течение суток. Чтобы обеспечить разумный баланс между сохранением новизны содержимого и нагрузкой на сервер, можно было бы предусмотреть хранение этой страницы в кэше не более пяти минут.

Развивая пример с двумя списками, создаваемыми двумя отдельными представлениями, можно было бы обеспечить их раздельное кэширование, применив декоратор.

```
from django.views.decorators.cache import cache_page

@cache_page(24 * 60 * 60)
def top_stories_yesterday(request):
    # ... извлекает статьи и возвращает HttpResponse

@cache_page(5 * 60)
def top_stories_today(request):
    # ... извлекает статьи и возвращает HttpResponse
```

Декоратор `cache_page` принимает единственный аргумент – продолжительность интервала времени в секундах, в течение которого страница должна находиться в кэше. Это все, что вам требуется для реализации такого механизма кэширования.

Декораторы кэширования представлений опираются на тот факт, что все представления Django принимают объект `HttpRequest` и возвращают объект `HttpResponse`. Первый они используют, чтобы определить запрашиваемый URL, – кэшируемые данные сохраняются в виде пар ключ-значение, где ключом является URL. Второй они используют для установки в ответе заголовков HTTP, имеющих отношение к кэшированию.

Управление заголовками, имеющими отношение к кэшированию

До настоящего момента мы рассматривали, что *вы* должны сделать на *своем* сервере, чтобы определить, как часто должна производиться регенерация данных в кэше. С практической точки зрения, кэширование включает в себя обмен определенной информацией между серве-

ром и клиентами, выполняющими подключение к нему (включая внешние кэширующие серверы, которыми вы не можете управлять). Эта информация формируется из специальных заголовков, которые называются заголовками «управления кэшированием» и передаются в составе ваших ответов HTTP.

Наиболее типичную из дополнительных форм управления кэшированием в платформе Django реализует декоратор «никогда не кэшировать».

```
from django.views.decorators.cache import never_cache

@never_cache
def top_stories_this_second(request):
    # ... нам необходимо, чтобы эту страницу никто не кэшировал
```

Этот декоратор предписывает всем получателям не сохранять данную страницу в кэше. Страница не будет сохраняться в кэше, если получатели соблюдают положения стандарта (RFC 2616). Фактически декоратор never_cache является оберткой вокруг более мощного и гибкого инструмента, связанного с кэшированием, предлагаемого платформой Django: django.views.decorators.cache.cache_control.

Декоратор cache_control модифицирует заголовок Cache-control в объекте HttpResponse с целью сообщить веб-клиентам и нижележащим кэширующим серверам о вашей политике кэширования. Вы можете передавать декоратору любое из шести логических значений (public, private, no_cache, no_transform, must_revalidate, proxy_revalidate) и два целочисленных значения (max_age, s_maxage).

Например, если вам необходимо вынудить клиентов и нижележащие кэширующие серверы «проверять», была ли обновлена ваша страница, даже если время хранения кэшированной версии страницы, которую они получили, еще не истекло, вы можете задекорировать свою функцию представления, как показано ниже:

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True)
def revalidate_me(request):
    # ...
```

Большинство сайтов едва ли нуждается в более точном управлении параметрами кэширования,чем то, которое обеспечивает декоратор cache_control. Но если подобная потребность действительно возникает, то у вас уже имеется необходимая функциональность и вам не придется вручную изменять заголовки в объекте HttpResponse.

Кроме того, платформа Django обеспечивает возможность управления заголовком Vary HTTP. Обычно при кэшировании содержимого в качестве ключа используется адрес URL. Однако у вас могут иметься дополнительные факторы, определяющие, какое содержимое должно возвращаться для данного адреса URL. Например, зарегистрированный и анонимный пользователи могут получать разные страницы, или от-

вет может зависеть от типа броузера у пользователя или от языковых настроек. Все эти факторы передаются серверу в виде заголовков HTTP при запросе страницы. Заголовок «Vary» в ответе позволяет точно указать, какие из заголовков в запросе оказывают влияние на содержимое.

Например, если для одного и того же адреса URL возвращается разное содержимое в зависимости от значения заголовка Accept-Language, вы можете потребовать от механизма кэширования учитывать этот заголовок.

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers("Accept-Language")
def localized_view(request):
    # ...
```

Поскольку в практике часто изменяется заголовок «Cookie», для удобства был создан простой декоратор vary_on_cookie.

Кэширование объектов

Выше рассматривались параметры, управляющие кэшированием *страниц* – всех страниц сайта, в случае использования механизма кэширования всего сайта, и отдельных страниц (представлений) при включении механизма кэширования для отдельных представлений.

Эти решения чрезвычайно просты в реализации. Однако в некоторых ситуациях инфраструктура кэширования может использоваться для сохранения отдельных элементов данных.

Допустим, что у вас имеется сайт, работающий с высокой нагрузкой, на каждой странице которого размещается некоторый блок информации, получаемой в результате дорогостоящих вычислений, например это могут быть результаты обработки огромного периодически обновляемого файла. Остальное содержимое страниц генерируется очень быстро, а поскольку этот блок информации отображается на многих страницах, есть смысл использовать механизм кэширования объектов.

Кэш объектов Django – это, в действительности, простое хранилище пар ключ/значение, позволяющее определять максимальное время хранения и дающее возможность сохранять и извлекать произвольные объекты. Благодаря этому вы можете сосредоточить свое внимание на тех из них, создание которых требует значительных вычислительных ресурсов. Ниже приводится фрагмент с нашим воображаемым примером, где пока отсутствует кэширование.

```
def stats_from_log(request, stat_name):
    logfile = file("/var/log/imaginary.log")
    stat_list = [line for line in logfile if line.startswith(stat_name)]
    # ... отображение шаблона, в котором выводится список stat_list
```

Несмотря на то, что генератор списков в третьей строке выглядит ничем не примечательным, тем не менее, обработка с его помощью боль-

ших файлов журналов может выполняться достаточно продолжительное время. Нам требуется избежать создания списка stat_list при каждом запросе. Решить эту задачу нам помогут методы cache.get и cache.set из модуля django.core.cache.

```
from django.core.cache import cache

def stats_from_log(request, stat_name):
    stat_list = cache.get(stat_name)
    if stat_list == None:
        logfile = file("/var/log/imaginary.log")
        stat_list = [line for line in logfile if line.startswith(stat_name)]
        cache.set(stat_name, stat_list, 60)
    # ... отображение шаблона, в котором выводится список stat_list
```

Метод cache.get возвращает значение (объект) из кэша, соответствующее заданному ключу, при условии, что время его хранения в кэше еще не истекло; в противном случае метод cache.get возвращает значение None, а элемент удаляется из кэша.

Метод cache.set принимает ключ (строку), значение (любое значение, которое может обработать модуль pickle из стандартной библиотеки языка Python) и необязательное время хранения объекта в кэше (в секундах). Если время не указано, будет использоваться значение параметра CACHE_BACKEND. Подробнее о параметре CACHE_BACKEND рассказывается ниже.

Кроме того, существует еще метод get_many, который принимает список ключей и возвращает словарь, отображающий эти ключи в их значения (возможно, продолжающие оставаться в кэше). Последнее замечание, – на тот случай, если вы сами этого не заметили: механизм кэширования объектов *никак* не связан с механизмом промежуточной обработки – мы просто импортировали django.core.cache и не предлагали изменить какие-либо настройки или добавить какой-либо промежуточный обработчик.

Тег шаблонов cache

Платформа Django предоставляет еще один способ кэширования: тег шаблонов cache. Этот тег обеспечивает возможность использовать механизм кэширования объектов внутри шаблона, не изменяя реализацию функции представления. Одним разработчикам не нравится идея появления таких артефактов оптимизации, как кэширование, на уровне отображения, хотя другие находят такую возможность целесообразной.

Допустим, к примеру, что у нас имеется шаблон, отображающий информацию в виде длинного списка элементов, а процесс создания такого списка требует существенных затрат вычислительных ресурсов. Допустим также, что остальная информация в этой странице, за исключением списка, изменяется при каждой ее загрузке, и потому простое

кэширование страницы не даст никаких преимуществ, а сам список требуется обновлять не чаще, чем один раз в пять минут. Поскольку «стоимость» вывода списка обусловлена наличием цикла внутри шаблона и вызовом дорогостоящего метода внутри этого цикла, нам не за что зацепиться в функции представления или в модели, чтобы решить эту проблему. Однако с помощью тега шаблонов cache мы можем задействовать механизм кэширования там, где это необходимо.

```
{% load cache %}
... Различные некэшированные части страницы ...
{% cache 300 list_of_stuff %}
    {% for item in really_long_list_of_items %}
        {{ item.do_expensive_rendering_step }}
    {% endfor %}
{% endcache %}
... Другие некэшированные части страницы ...
```

В этом примере кэшируются все данные, выводимые внутри цикла for. Тег cache принимает два аргумента: продолжительность интервала времени хранения в кэше в секундах и ключ содержимого в кэше.

В некоторых случаях статический ключ не может использоваться для помещения данных в кэш. Например, когда сайт поддерживает возможность локализации и отображаемые данные зависят от языковых настроек броузера пользователя, было бы желательно, чтобы ключ кэша отражал этот факт. К счастью, тег cache имеет третий необязательный параметр, предназначенный для использования как раз в таких ситуациях. Этот параметр является именем переменной шаблона, значение которой объединяется со статической частью имени ключа (в предыдущем примере – list_of_stuff) для создания полного имени.

Для учета того, что содержимое списка list_of_stuff зависит от языковых настроек, определение тега cache может выглядеть, как показано ниже:

```
{% cache 300 list_of_stuff LANGUAGE_CODE %}
```

Примечание

В этом последнем примере предполагается, что вы передаете своим шаблонам RequestContext, который добавляет дополнительные переменные в контекст шаблона, исходя из настроек процессора контекста. Процессор интернационализации django.core.context_processors.i18n активирован по умолчанию и предоставляет переменную LANGUAGE_CODE. За дополнительной информацией о процессорах контекста обращайтесь к главе 6 «Шаблоны и обработка форм».

Типы механизмов кэширования

В предыдущем введении в инфраструктуру кэширования платформы Django мы познакомились с механизмом «locmem». Ниже приводится полный список всех доступных механизмов:

- dummy. Используется только в процессе разработки. В действительности этот механизм не выполняет кэширование, но позволяет не трогать остальные настройки механизма кэширования, благодаря чему приложения будут корректно кэшироваться на рабочем сайте (где используется один из следующих ниже типов механизмов).
- locmem. Надежный кэш, располагающийся в оперативной памяти, который можно безопасно использовать сразу несколькими процессами. Используется по умолчанию.
- file. Кэш располагается в файловой системе.
- db. Кэш располагается в базе данных (требует создания специальной таблицы).
- memcached. Высокопроизводительный, распределенный кэш в памяти – наиболее мощный механизм.

Параметр CACHE_BACKEND принимает значение в форме URL, которое начинается с имени типа механизма, вслед за которым идут символ двоеточия и два символа слеша (три символа слеша – в случае использования механизма file). Механизмы, используемые при разработке, dummy и locmem, не имеют дополнительных аргументов. Настройка механизмов file, db и memcached описывается ниже.

Параметр CACHE_BACKEND принимает также три необязательных аргумента.

- max_entries: Максимальное число уникальных записей в кэше, значение по умолчанию – 300. Не забывайте, что большая доля нагрузки на сервер обычно обусловливается небольшим числом элементов, поэтому, чтобы обеспечить высокую производительность, вовсе не требуется сохранять в кэше *все подряд*. А из-за ограниченного времени хранения данных в кэше храниться в нем будут преимущественно наиболее часто используемые элементы.
- cull_percentage: Не совсем точное название. В действительности этот параметр определяет не процент, а долю записей в кэше, которая должна быть удалена по достижении предела max_entries. Значение по умолчанию 3 означает, что при каждом заполнении кэша будет удаляться 1/3 наиболее «старых» записей.
- timeout: Предельное время хранения содержимого в кэше в секундах. Значение по умолчанию – 300 секунд (пять минут). Это значение используется не только для определения момента времени, когда данные должны удаляться из кэша, но также при создании различных заголовков HTTP, сообщающих веб-клиентам информацию, имеющую отношение к кэшированию.

Эти параметры указываются в стиле аргументов URL, как показано ниже:

```
CACHE_BACKEND = "locmem://?max_entries=1000&cull_percentage=4&timeout=60"
```

Такое значение сообщает платформе Django, что для организации кэша следует использовать оперативную память, хранить одновременно в кэше не более 1000 записей, при переполнении кэша удалять 1/4 часть записей и предельное время хранения составляет 60 секунд.

Файлы

Единственное, что требуется для работы механизма `file` – это каталог, доступный для записи процессу веб-сервера. Не забывайте указывать три символа слеша после двоеточия – первые два отмечают конец части URL, описывающей «схему», а третий указывает на абсолютный путь к каталогу (то есть путь должен начинаться от корневого каталога файловой системы). Подобно другим параметрам настройки Django, где указываются имена файлов, для разделения элементов пути здесь следует использовать символ слеша, – даже в операционной системе Windows.

```
CACHE_BACKEND = "file:///var/cache/django/mysite"
```

Конечно, в системе Windows этот параметр будет выглядеть, скорее всего, примерно так:

```
CACHE_BACKEND = "file:///C:/py/django/mysite/cache"
```

База данных

Чтобы использовать механизм кэширования, опирающийся на использование базы данных, необходимо создать в базе данных таблицу для кэша. Для этого можно выполнить следующую команду:

```
$ python manage.py createcachetable cache
```

Последний аргумент в этой команде – имя таблицы. Мы рекомендуем использовать имя `cache`, как это сделали мы, но вообще вы можете использовать любое имя по своему выбору. После создания таблицы параметр настройки механизма кэширования приобретает следующий вид:

```
CACHE_BACKEND = "db://cache/"
```

Это очень простая таблица, состоящая всего из трех столбцов: `cache_key` (первичный ключ в таблице), `value` (фактические данные, помещаемые в кэш) и `expires` (поле типа `datetime`, для повышения скорости работы Django устанавливает индекс на это поле).

Memcached

Механизм кэширования Memcached является наиболее мощным из тех, что предлагается платформой Django. Неудивительно, что он также является самым сложным в настройке. Однако, если он вам потребуется, эту сложность стоит преодолеть. Первоначально этот механизм был разработан для сайта *LiveJournal.com*, чтобы снизить нагрузку на их сервера баз данных при 20 миллионах просмотров стра-

ниц в день. С тех пор этот механизм также стал использоваться на *Wikipedia.org*, *Fotolog.com*, *Slashdot.org* и других сайтах, испытывающих высокую нагрузку. Домашняя страница проекта Memcached находится по адресу: *http://danga.com/memcached*.

Среди прочих возможностей механизма Memcached, перечисленных ниже, имеется возможность простого распределения кэша между несколькими серверами. Memcached – это «огромная хеш-таблица» – она используется подобно отображению ключ-значение, такому как словарь в языке Python, но при этом данные распределяются по указанным вами серверам.

Хотя механизм Memcached обладает более высокой производительностью, чем другие механизмы кэширования, представленные здесь, тем не менее, он остается обычным механизмом кэширования данных в памяти. Он не является объектной базой данных. Один из сборников ответов на часто задаваемые вопросы по Memcached на такие вопросы, как: «Насколько избыточен механизм Memcached?», «Как в нем обрабатываются ошибочные ситуации?» и «Как загрузить или удалить данные из кэша Memcached?» дает ответы: «Не избыточен. Такая возможность не предусмотрена. Возможность вам недоступна!». Надежным хранилищем информации служит ваша база данных, а механизм Memcached просто умеет работать с ней быстро. (За дополнительной информацией об истории создания и интереснейших особенностях архитектуры Memcached обращайтесь к статье по адресу *http://www.linuxjournal.com/article/7451*.)

Чтобы задействовать механизм Memcached, вам необходимы две вещи: собственно программное обеспечение и библиотеки языка Python, посредством которых Django сможет общаться с механизмом Memcached. Отыскать пакеты для вашего дистрибутива Linux или порты для вашей системы Mac OS X будет совсем несложно. Реализацию Memcached для Windows можно найти по адресу *http://splinedancer.com/memcached-win32*.

Затем на сервере, где работает приложение на платформе Django, вам необходимо обеспечить возможность взаимодействий с механизмом Memcached из программ на языке Python. Сделать это можно либо с помощью клиента `python-memcached` на языке Python (*http://tummy.com/Community/software/python-memcached*), либо использовав более быструю версию `cmemcache`, которая опирается на библиотеку, написанную на языке C (*http://gijsbert.org/cmemcache/*). Пакет `python-memcached` можно также загрузить и установить с помощью инструмента Easy Install.

Настройте свой сервер так, чтобы он автоматически запускал демон memcached во время загрузки. Демон не имеет конфигурационных файлов, поэтому все параметры передаются ему в командной строке при запуске. Следующая команда запускает memcached в режиме демона,

выделяет ему память объемом 2 Гбайта и предписывает принимать запросы на IP-адресе 10.0.1.1:

```
$ memcached -d -m 2048 -l 10.0.1.1
```

Если вам будет интересно ознакомиться с полным перечнем доступных параметров командной строки для memcached, обращайтесь к его странице справочного руководства man или к другой документации. В POSIX-совместимых системах вам следует поместить эту команду в сценарий начальной загрузки системы, а в системе Windows запуск демона следует оформить как службу.

Теперь, когда демон memcached запущен, необходимо сообщить платформе Django, что она может использовать его – с помощью параметра CACHE_BACKEND:

```
CACHE_BACKEND = "memcached://10.0.1.1:11211"
```

Платформа Django требует указывать номер порта. По умолчанию Memcached использует порт 11211, а так как мы не указывали номер порта в предыдущей команде, именно этот порт будет использоваться демоном Memcached. Если вы используете несколько серверов, отделяйте их друг от друга точкой с запятой.

```
CACHE_BACKEND = "memcached://10.0.1.1:11211;10.0.5.5:11211"
```

В качестве вывода можно сказать следующее: несмотря на то, что для настройки механизма Memcached требуется приложить немного больше усилий, чем в случае других механизмов, тем не менее, он остается простым механизмом и поэтому при корректной настройке ведет себя идентично другим механизмам.

Тестирование приложений на платформе Django

Ни у кого не вызывает сомнений, что наличие комплектов автоматизированных тестов для приложения обеспечивает дополнительные удобства. Это особенно верно в случае использования языков с динамической типизацией, таких как Python, которые не обеспечивают проверку типов на этапе компиляции.

Примечание

В этой главе предполагается, что вы уже свято верите в необходимость тестирования, и поэтому здесь рассматриваются вопросы *как*, а не *зачем*. Если вам нужны дополнительные обоснования этой необходимости, обращайтесь к другим источникам информации или посетите сайт *withdjango.com*.

Стандартная библиотека языка Python содержит два взаимодополняющих модуля поддержки тестирования – doctest и unittest, а кроме того, имеется множество популярных независимых инструментов. В этой главе, как и в платформе Django, основное внимание уделяется

двум встроенным системам, но если вам интересно побольше узнать об инструментах тестирования для языка Python, обращайтесь по адресу URL, который приводился выше.

Основное неудобство веб-приложений состоит в том, что тестировать их достаточно сложно. Дело также осложняется свойственной веб-приложениям разветвленной сетью взаимодействий, таких как соединения с базами данных, обработка запросов и ответов HTTP, рассылка писем по электронной почте и тому подобное. Однако есть и *хорошие* новости – поддержка тестирования в Django обеспечивает относительно простую возможность включения тестов в ваши проекты. Прежде чем погрузиться в особенности поддержки тестирования в платформе Django, познакомимся с некоторыми встроенными возможностями языка Python.

Основы доктестов

Доктест – это просто копия сеанса работы в интерактивной оболочке интерпретатора Python, включенная в **строку документирования** внутри модуля или функции. Модуль `doctest` используется, чтобы отыскать такие тесты внутри модуля, выполнить их и проверить полученные результаты. Краткий обзор строк документирования и порядок их использования даются в главе 1, «Практическое введение в Python для Django».

Например, ниже приводится упрощенная функция, для которой мы легко можем написать тест.

```
def double(x):
    return x * 2
```

Чтобы протестировать эту функцию вручную, мы могли бы в интерактивной оболочке интерпретатора выполнить такую команду:

```
>>> double(2)
4
```

Мы получили ожидаемый результат и объявили, что функция прошла тест. Чтобы добавить доктест в функцию, достаточно просто скопировать содержимое интерактивного сеанса в строку документирования.

```
def double(x):
    """
    >>> double(2)
    4
    """
    return x * 2
```

Когда эта функция будет тестироваться модулем `doctest`, он выполнит команду `double(2)`. Если в результате будет получено значение «4», тест будет считаться пройденным. В противном случае будет выведено сообщение об ошибке.

Модуль doctest автоматически определяет, какой текст не относится к тесту (например, обычный текст описания, не начинающийся с приглашения к вводу >>> и не следующий непосредственно за ним), поэтому мы без всяких ограничений можем добавлять дополнительное описание.

```
def double(x):
    """
    Мы предполагаем, что эта функция должна возвращать
    удвоенное значение аргумента.
    >>> double(2)
    4
    """
    return x * 2
```

Основы модульного тестирования

Модуль unittest дополняет возможности модуля doctest, реализуя иной подход к тестированию. Он является адаптацией платформы тестирования JUnit для языка Java, который в свою очередь корнями уходит в оригинальную реализацию системы тестирования для языка Smalltalk. Типичные простые тесты для использования совместно с модулем unittest выглядят примерно так:

```
import unittest

class IntegerArithmeticTestCase(unittest.TestCase):
    def testAdd(self):
        self.assertEquals(1 + 2, 3)
    def testMultiply(self):
        self.assertEquals(5 * 8, 40)

if __name__ == '__main__':
    unittest.main()
```

Этот пример представляет собой законченный сценарий – при его выполнении в виде самостоятельной программы он запускает свой набор тестов. Это происходит благодаря вызову функции unittest.main(), которая отыскивает все подклассы класса unittest.TestCase и вызывает все методы этих классов, имена которых начинаются со слова test.

Запуск тестов

Тесты в платформе Django могут запускаться командой:

```
./manage.py test
```

Платформа Django автоматически отыщет тесты (любого вида) в файлах models.py всех приложений, перечисленных в параметре INSTALLED_APPS. У вас имеется возможность сузить круг поиска с помощью дополнительных аргументов команды test, определяющих отдельные

приложения или даже отдельные модели внутри приложений, – например manage.py test blog или manage.py test blog.Post.

Кроме того, команда test попытается отыскать все модульные тесты во всех файлах с именем test.py, находящиеся внутри каталогов приложений (в подкаталогах, находящихся на том же уровне, что и файлы models.py). Благодаря этому вы можете сохранять свои модульные тесты в любом из двух местоположений по своему желанию.

Тестирование моделей

Обычно модели тестируются с помощью доктестов, так как платформа Django автоматически отыскивает и выполняет доктесты во всех установленных приложениях, когда вы запускаете команду manage.py test. Если у вас имеется некоторая базовая модель, состоящая исключительно из полей данных, то в ней в общем-то нечего тестировать. В этом случае модель является простым представлением данных, которое обслуживается хорошо протестированной внутренней логикой платформы Django. Однако, как только вы начнете добавлять в модель методы, тут же возникает необходимость тестировать логику их работы.

Например, предположим, что у вас имеется модель Person, которая включает поле birthdate с датой дня рождения, и метод, вычисляющий возраст человека на определенный день. Реализация такой модели могла бы выглядеть, как показано ниже:

```
from django.db import models

class Person(models.Model):
    first = models.CharField(max_length=100)
    last = models.CharField(max_length=100)
    birthdate = models.DateField()

    def __unicode__(self):
        return "%s %s" % (self.first, self.last)

    def age_on_date(self, date):
        if date < self.birthdate:
            return 0
        return (date - self.birthdate).days / 365
```

Программный код, представленный в методе age_on_date, подвержен печально известной ошибке, когда при граничных условиях (например, при тестировании даты дня рождения) могут возвращаться неверные результаты. Используя доктесты, мы можем защититься от этой и других ошибок.

Если бы мы вручную проверяли работу нашего метода age_on_date, мы запустили бы интерактивную оболочку интерпретатора Python, создали бы пример объекта и вызвали бы его метод, как показано ниже:

```
>>> from datetime import date
>>> p = Person(firstname="Jeff", lastname="Forcier", city="Jersey City",
```

```
... state="NJ", birthdate=date(1982, 7, 15))
>>> p.age_on_date(date(2008, 8, 10))
26
>>> p.age_on_date(date(1950, 1, 1))
0
>>> p.age_on_date(p.birthdate)
0
```

Теперь, когда вы уже знаете о существовании доктестов, мы можем просто взять текст, полученный в ходе сеанса работы с интерпретатором, и поместить его в строку документирования метода `age_on_date`**, после чего реализация метода обретает следующий вид:**

```
def age_on_date(self, date):
    """
    Возвращает целое число - возраст человека в годах на указанную дату.
    >>> from datetime import date
    >>> p = Person(firstname="Jeff", lastname="Forcier",
    ... city="Jersey City", state="NJ", birthdate=date(1982, 7, 15))
    >>> p.age_on_date(date(2008, 8, 10))
    26
    >>> p.age_on_date(date(1950, 1, 1))
    0
    >>> p.age_on_date(p.birthdate)
    0
    """
    if date < self.birthdate:
        return 0
    return (date - self.birthdate).days / 365
```

Наконец, мы можем использовать упомянутую выше команду `manage.py` **для выполнения этого теста:**

```
user/opt/code/myproject $ ./manage.py test myapp
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table myapp_person
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
.
----------------------------------------------------------------------
Ran 1 test in 0.003s
OK

Destroying test database...
```

Конечно, это слишком большой объем вывода для одного маленького теста, но в самом его конце, после тестирования полной иерархии моделей, будут выведены одна или две строки из точек, в которой периодически могут появляться символы E и F – в случае выявления ошибок, обнаруженных при прохождении тестов.

Наконец, обратите внимание: хотя в большинстве случаев доктесты способны удовлетворить практически любые ваши потребности в тестировании, тем не менее, не ленитесь создавать модульные тесты для проверки сложной управляющей логики или взаимоотношений между моделями. Если вы только начинаете знакомиться с миром тестирования, вам потребуется некоторое время, чтобы выяснить, что и когда можно использовать, – но не сдавайтесь!

Тестирование всего веб-приложения в целом

Полное тестирование веб-приложения сверху донизу – далеко не простая задача и она не может быть автоматизирована на все 100 процентов с помощью одних и тех же сценариев, так как каждое веб-приложение имеет свои отличительные особенности. Однако существует ряд инструментов, которые могут оказаться как нельзя кстати.

Для начала следует обратить внимание на инструмент, встроенный непосредственно в саму платформу Django и ко времени написания этих строк еще считавшийся новинкой. Он называется «тестовым клиентом Django», а его описание можно найти на официальном веб-сайте проекта Django по адресу: *http://www.djangoproject.com/documentation/testing/#testing-tools*. Тестовый клиент обеспечивает простой способ смоделировать типичный цикл запрос-ответ и проверить выполнение определенных условий.

Когда вы обнаружите, что вам необходим больший контроль над выполнением тестов, чем способен дать тестовый клиент, тогда можно попробовать использовать более старый инструмент Twill, обладающий более широкими возможностями, который вы найдете по адресу: *http://twill.idyll.org/*. Как и тестовый клиент Django, он опирается на использование командной строки и, несмотря на простоту в использовании, является мощной и типичной «питонической» библиотекой.

Еще один инструмент тестирования, вызвавший волну интереса совсем недавно, – это Selenium (*http://selenium.openqa.org/*). В отличие от первых двух, этот инструмент тестирования основан на коде HTML/JavaScript, созданном специально для тестирования веб-приложений с позиций броузера. Он поддерживает работу с основными типами броузеров на большинстве платформ и, благодаря тому, что опирается на программный код JavaScript, может использоваться для тестирования функциональных особенностей реализации технологии Ajax. Весь программный код приложения делится на три части по режиму работы: Selenium Core, Selenium RC (Remote Control – удаленное управле-

ние) и Selenium IDE (Integrated Development Environment – интегрированная среда разработки).

Selenium Core (*http://selenium-core.openqa.org/*) – это основа, или ядро инструмента тестирования (как вручную, так и автоматически) веб-приложений. Некоторые говорят, что эта часть инструмента Selenium выполняется в «режиме бота». Это «рабочая лошадка» всего набора. В дополнение к возможности протестировать функциональность веб-приложения ядро в состоянии протестировать его совместимость с броузером.

Selenium RC (*http://selenium-rc.openqa.org/*) дает пользователям возможность создавать полноценные автоматизированные тесты на различных языках программирования. Благодаря этому компоненту вы можете писать свои тестовые приложения, управляющие ядром Selenium Core. Этот компонент Selenium можно представить себе, как промежуточный слой программного обеспечения поверх Selenium Core, работающий, если хотите, «в режиме программного кода».

Замечательным инструментом, который позволит начать использовать Selenium, является интегрированная среда разработки (*http://selenium-ide.openqa.org/*). Она написана как расширение для броузера Firefox и представляет собой полноценную среду разработки, позволяющую записывать и воспроизводить сеансы работы с веб-приложениями как тесты. Она также может выводить тесты на любом языке программирования, поддерживаемом пакетом Selenium RC, благодаря чему вы можете дополнять и модифицировать эти тесты. Вы можете устанавливать точки останова и производить пошаговое выполнение тестов. Поскольку эта среда разработки написана в виде расширения для броузера Firefox, часто задают вопрос – не существует ли аналогичного расширения для Internet Explorer (IE). Ответ на этот вопрос – нет. Тем не менее, «режим записи», имеющийся в среде разработки, позволяет запускать записанные сеансы на воспроизведение под управлением IE с помощью Selenium Core.

Помимо этих трех инструментов – тестовый клиент Django, Twill и Selenium, вы можете найти немало информации о тестировании веб-приложений по адресу *http://www.awaretek.com/tutorials.html#test*, где приводятся ссылки на другие инструменты.

Тестирование программного кода самой платформы Django

Сама платформа Django содержит обширнейший набор тестов. Как правило, каждое исправление ошибки сопровождается созданием **регрессионного теста**, который гарантирует, что исправленная ошибка не останется незамеченной. При добавлении новых функциональных возможностей также обычно создаются тесты, позволяющие убедиться, что они работают так, как ожидалось.

Вы можете попробовать выполнить эти тесты самостоятельно. Это может потребоваться, например, когда возникают проблемы с запуском платформы Django на редко используемой платформе или в необычной конфигурации. Сначала желательно проверить свой собственный программный код; тем не менее, вполне возможно, что вы столкнулись с необычной внутренней ошибкой, которая ранее не выявлялась. Выявление ошибок при прохождении внутреннего набора тестов дает вам право составить отчет об ошибке, к которому разработчики отнесутся достаточно серьезно.

Запустить набор тестов достаточно просто, правда есть одно незначительное препятствие: необходимо указать файл с настройками, чтобы можно было определить, как создавать базу данных для тестов. Это может быть файл с настройками любого из проектов, можно также создать фиктивный проект (то есть проект без приложений в нем) и определить в нем значения параметров DATABASE_*, в файле settings.py.

Сценарий, запускающий тесты, находится в подкаталоге tests, расположенном в каталоге установки Django. (Не путайте его с пакетом test, который является составной частью Django.) Команда запуска тестов выглядит, как показано ниже:

```
$ tests/runtests.py -settings=mydummyproject.settings
```

Тестирование проходит по большей части без вывода информации – выводится только информация об обнаруженных ошибках. Поскольку на прохождение всех тестов в наборе может потребоваться достаточно продолжительное время, вы можете попробовать получить больше информации в ходе выполнения тестов. Команда runtests.py принимает ключ -v, повышающий ее информативность. С ключом -v1 в ходе выполнения тестов команда выводит информацию, как показано ниже:

```
.................................................E...EE...
```

Символ E свидетельствует о том, что была обнаружена ошибка. Вслед за этой строкой следует подробный отчет о природе возникших ошибок, благодаря которому вы сможете определить, вызваны ли они вашими настройками или проблема действительно кроется в самой платформе Django.

С ключом -v2 вывод приобретает форму длинного перечня импортированных моделей, вслед за которыми следуют сообщения, подробно описывающие создание и удаление тестовой базы данных и таблиц в ней (многоточия в следующем примере заменяют строки, которые мы удалили, чтобы сократить листинг).

```
Importing model basic
Importing model choices
Importing model custom_columns
Importing model custom_managers
Importing model custom_methods
Importing model custom_pk
```

```
Importing model empty
...
Creating test database...
Processing contenttypes.ContentType model
Creating table django_content_type
Processing auth.Message model
Creating table auth_message
Processing auth.Group model
Creating table auth_group
Processing auth.User model
Creating table auth_user
...
```

Наличие ошибок при прохождении тестов совершенно необязательно означает, что вы обнаружили проблемы в платформе Django, – если вы в этом не уверены, для начала стоит написать в почтовую рассылку, подробно описав свою конфигурацию и вывод с сообщением об ошибке.

В заключение

В этой главе рассматривалось множество дополнительных тем, и хотелось бы надеяться, что в совокупности с темами главы 11 нам удалось показать глубину возможностей разработки веб-приложений при использовании платформы Django. Конечно, эти темы являются лишь примером того, что возможно: разработка веб-приложений, как и многие другие компьютерные дисциплины, не является самостоятельной и изолированной областью человеческих знаний, она тесно переплетается со многими другими областями, так же, как сам язык программирования Python, который можно использовать для решения широкого круга задач и реализации различных технологий.

В этом месте вы уже закончили чтение книги – поздравляем! Остались лишь приложения, но они, как и эти две главы, являются важной частью книги, охватывая различные темы – начиная от использования командной строки и описания процедуры установки и развертывания платформы Django и заканчивая списком внешних ресурсов и средств разработки.

В заключение, возможно, было бы полезно вернуться назад и перечитать (или хотя бы бегло просмотреть) начало книги. Теперь, когда вы уже познакомились со всеми темами, которые мы хотели представить, ранние примеры программного кода и объяснения могут обрести для вас дополнительный смысл. Безусловно, это справедливо для любой специальной книги, а не только для этой.

V

Приложения

A. Основы командной строки
B. Установка и запуск Django
C. Инструменты разработки для платформы Django
D. Поиск, оценка и использование приложений на платформе Django
E. Django и Google App Engine
F. Участие в проекте Django

A

Основы командной строки

Огромное большинство веб-серверов (не говоря уже о серверах электронной почты, файловых серверах и т. д.) работают под управлением POSIX-совместимых операционных систем, таких как Linux, FreeBSD и других UNIX-подобных систем. И веб-серверы, опирающиеся на использование платформы Django, не являются исключением – большая часть разработчиков Django и большая часть пользователей используют эту платформу именно на таких машинах. Если ранее вам не приходилось сталкиваться с интерфейсом командной строки, типичным для таких систем, в этом приложении вы получите краткий обзор этого интерфейса, после чего остальная часть книги приобретет для вас больше смысла.

Тем, кто пришел из мира Windows, это приложение, возможно, даст не очень много практических знаний, но, тем не менее, мы предлагаем прочитать его (или хотя бы бегло просмотреть), потому что в любом случае есть вероятность, что приобретенные здесь знания пригодятся вам в будущем. Кроме того, по общему мнению, чем большим числом языков программирования, платформ и технологий владеет программист, тем эффективнее он может использовать знакомые ему инструменты и тем выше его способности к освоению новых инструментов.

Если у вас появится желание попрактиковаться с описываемыми здесь командами в операционной системе Windows, мы рекомендуем установить Cygwin – командную оболочку для Windows, напоминающую командную оболочку Linux. Она содержит слой эмуляции и набор инструментов командной строки, хорошо знакомых пользователям UNIX, часть которых будет демонстрироваться в этом приложении. Однако эта оболочка не предназначена для того, чтобы превратить ваш персональный компьютер в сервер. Дополнительную информацию

о продукте Cygwin, как и сам продукт, вы найдете на сайте *http://cygwin.com*.

Если вы пользуетесь Mac OS X, можете считать, что вам повезло. Система MacOS X является одним из ответвлений BSD (Berkeley Software Distribution) UNIX и обеспечивает вашему компьютеру большую часть функциональных возможностей полноценного сервера. Чтобы поэкспериментировать с интерфейсом командной строки, в этой системе достаточно просто запустить приложение Terminal (найти его можно в /Applications/Utilities). Начиная с этого момента мы будем предполагать, что у вас имеется доступ какой-либо командной оболочке UNIX, в которой вы сможете выполнять описываемые команды.

Ввод «команды» в «командную строку»

Управление UNIX-подобными операционными системами осуществляется не с помощью графического интерфейса, когда выполняются щелчки мышью на кнопках и заполняются поля ввода, а с помощью **командных интерпретаторов**, или **командных оболочек**, когда строка приглашения принимает команды и выполняет по очереди. Как программист вы наверняка знакомы с такими понятиями, как печать строки, вызов функции с некоторыми параметрами и т. д. Командная строка представляет собой практически то же самое.

Ниже приводится простой пример, в котором выводится содержимое текущего каталога (в Windows каталоги называются папками), затем содержимое подкаталога и удаляется файл в этом подкаталоге. Обратите внимание на символ $ – это приглашение к вводу; он служит признаком, отличающим строки, в которых вводятся команды, от строк, где выводятся результаты работы этих команд.

В других командных оболочках могут использоваться другие символы, не только $. Вы можете увидеть символ > или %. (В интерактивной оболочке интерпретатора Python в качестве приглашения к вводу используется последовательность символов >>>.) В дополнение к символу (или к символам), находящимся в строке непосредственно перед вводом пользователя, во многих случаях приглашения к вводу содержат дополнительную информацию, например имя текущего пользователя, имя хоста или каталога (как в некоторых примерах этой книги, где строка приглашения к вводу имеет вид `user@example $`).

Ниже следует короткий пример, где сначала выводится содержимое текущего каталога, а затем выполняется удаление файла в подкаталоге.

```
$ ls
documents code temp
$ ls documents
test.py
$ rm documents/test.py
$
```

Обе команды, использованные в этом примере, являются программами, или двоичными файлами, находящимися где-то в пути поиска выполняемых программ. (Подробнее о путях поиска рассказывается в следующем разделе.) Даже при том, что теоретически программы могут выполнять самые разные действия, тем не менее, имеются определенные стандарты, определяющие порядок передачи ключей и аргументов. Традиционно команды UNIX состоят из трех частей: имени команды; ключей, управляющих поведением команды; и аргументов, которые определяют выполняемые подкоманды, имена файлов, которые следует обработать, и т. д.

Если взять последнюю команду из предыдущего примера, то rm — это имя программы (название rm происходит от английского «remove» — удалить), а documents/test.py — это аргумент, то есть файл, который требуется удалить. Если бы нам потребовалось удалить весь каталог, мы могли бы передать команде дополнительные ключи, управляющие ее поведением, как показано ниже:

```
$ ls temp
tempfile1 tempfile2
$ rm temp
rm: невозможно удалить 'temp': Это каталог
$ rm --help
Использование: rm [КЛЮЧ]... ФАЙЛ...
Удаляет (ссылки на) ФАЙЛ(ы).

    -f, --force        игнорировать несуществующие файлы, ни о чем не спрашивать
    -i                 запрашивать подтверждение перед каждым удалением
    -I                 запрашивать подтверждение один раз перед удалением
                       более чем трех файлов или перед рекурсивным удалением.
                       Не так назойливо, как -i, но все же предоставляет
                       защиту от большинства ошибок.
    --interactive[=КОГДА]  запрашивать подтверждение, КОГДА указано:
                           never (никогда), once (-I, один раз)
                           или always (-i, всегда). Если КОГДА не задано,
                           запрашивать всегда.
    --one-file-system  при рекурсивном удалении иерархии пропускать
                       все каталоги, находящиеся не на той же файловой
                       системе, что и соответствующий аргумент
                       командной строки
    --no-preserve-root не обрабатывать '/' особо
    --preserve-root    отказываться рекурсивно обрабатывать '/'
                       (по умолчанию)
    -r, -R, --recursive рекурсивно удалять каталоги и их содержимое
    -v, --verbose      пояснять производимые действия
    --help             показать эту справку и выйти
    --version          показать информацию о версии и выйти
$ rm -rf temp
$
```

В общем случае команда rm не может удалять каталоги, именно поэтому мы оказались не в состоянии удалить каталог. Однако, если ей передать ключи -r и -f, объединив их в одну строку для удобства (подробнее об этом рассказывается ниже, в разделе «Ключи и аргументы»), то команда rm рекурсивно удалит вложенные подкаталоги, не задавая никаких вопросов, и затем без каких-либо проблем удалит каталог temp.

Как видно из предыдущего примера, программы обычно содержат встроенную в них справочную информацию о допустимых ключах и аргументах. Практически все программы в операционной системе UNIX принимают ключ --help или -h, при использовании которого они выводят справочное сообщение. Обычно эти сообщения содержат достаточно информации об использовании программы – как для начинающего, так и для опытного пользователя.

Имена программ UNIX

Имена многих, если не большинства, программ UNIX выглядят несколько странно, например rm, ls, sed и т. д. Отчасти такие имена сложились по историческим причинам (клавиатуры и дисплеи в прежние времена были сильно медлительнее современных), а отчасти из соображений экономии на вводе – в средах, целиком или в значительной степени опирающихся на использование клавиатуры, можно сделать гораздо больше, если ввод команд будет занимать меньше времени.

Иногда сокращенные имена выглядят достаточно очевидными, например rm («remove» – удалить) и ls («list» – перечислить). Иногда менее очевидными, например sed – это сокращенно от «stream editor» (редактор потока). Многие имена, особенно имена сравнительно новых программ, представляют собой акронимы, составленные из других акронимов, например популярная программа gcc – это сокращенно «GNU C compiler»[1] (компилятор C GNU), где GNU – это название проекта «GNU's Not Unix».

Вообще команды UNIX часто становятся устойчивыми акронимами – как только вы узнаете, для каких целей используется та или иная команда, фактический смысл акронима перестает иметь большое значение, так как сокращенная форма приобретает отдельный смысл.

---

[1] На сайте проекта GCC и в википедии дается иное толкование: «GNU Compiler Collection» (коллекция компиляторов GNU), и это ближе к истине! – *Прим. перев.*

Примечание

Если вы пытаетесь выполнять команды вместе с нами, информация, выводимая у вас, может несколько отличаться от той, что показана здесь, вследствие различий между версиями систем UNIX – обычно программы имеют несколько отличающиеся реализации в зависимости от платформы. Базовые утилиты, такие как rm и ls, также могут иметь некоторые различия, хотя, как правило, некоторое подмножество ключей является общим для всех систем.

Если встроенной справки недостаточно и вам необходимо более подробное разъяснение о некотором параметре, существует еще система man (сокращенно от «[user] manual» – руководство пользователя), которая содержит полный комплект информации о каждой команде, часто с более подробными описаниями аргументов и/или с примерами использования. Сама команда man также является программой, как и все остальные, и обычно принимает единственный аргумент – имя программы, чью «страницу руководства» вы хотели бы получить. Начинающие пользователи часто вызывают команду man man, которая, как нетрудно догадаться, выводит страницу справочного руководства для самой команды man.

Так или иначе, UNIX-подобные системы часто ориентированы на тех, кто занимается самообразованием, а так как вполне очевидно, что вы не ленитесь читать, авторы уверены, что вы справитесь с этим! А если говорить серьезно, вам пойдет на пользу, если вы выработаете у себя привычку сначала попытаться найти ответ самому и только потом обращаться за помощью к другим.

Ключи и аргументы

Выше мы представили ключи и аргументы программы как два различных аспекта спецификации, хотя в действительности это не совсем так. По сути ключи и аргументы – это просто длинная строка, передаваемая программе, которая может интерпретироваться программой по ее собственному усмотрению. Вследствие этого стандарты, представленные здесь, – это просто стандарты, а на практике часто можно встретить их вариации, образовавшиеся в соответствии с тем, насколько строго автор программы придерживался установленных норм.

Вообще говоря, все аргументы или ключи программы отделяются пробелами, при этом ключи обычно предваряются символом дефиса (-) и располагаются перед аргументами, которые не имеют префикса. Некоторые программы также принимают так называемые длинные ключи, когда в качестве префикса используются два символа дефиса и имя ключа состоит более чем из одного символа, например ключ --help, упоминавшийся выше.

```
$ rm --help
Использование: rm [КЛЮЧ]... ФАЙЛ...
```

Вот стандарт: имя программы/утилиты, за которым следует ноль или более ключей любого типа, за которыми следует ноль или более аргументов (так как некоторые программы вообще могут не принимать аргументы). Ключи могут указываться отдельно друг от друга, разделенные пробелами:

```
$ rm -r -f temp
```

или объединенные в единую строку параметров, чтобы сократить объем ввода, как показано в следующем примере:

```
$ rm -rf temp
```

Обратите внимание, что таким способом нельзя объединять длинные и короткие ключи, так как в этом нет никакого смысла, но они могут использоваться вперемешку:

```
$ rm -rf --verbose temp
```

Ключи могут иметь собственные аргументы в зависимости от вызываемой команды. Например, программа head по умолчанию возвращает только первые десять строк из указанного файла, однако это число можно изменить с помощью аргумента ключа -n, как показано в следующем примере, где команда возвращает первые пять строк из файла myfile.txt:

```
$ head -n 5 myfile.txt
```

По умолчанию (без ключа -n) команда head отображает первые десять строк. Обратите внимание, что аналогично объединению нескольких ключей в одну строку аргументы ключей не обязательно отделяются от самих ключей пробелом, а могут объединяться в одну строку с ними.

```
$ head -n5 myfile.txt
```

Наконец, несмотря на то, что программы UNIX стремятся принимать ключи и аргументы в порядке <program> <options> <arguments>, многие программы Linux дают большую свободу выбора в порядке следования элементов команды, как показано ниже:

```
$ head myfile.txt -n5
```

или

```
$ rm -rf temp --verbose
```

Хотя такое поведение программ Linux довольно удобно (так как при этом пользователю предоставляется возможность добавить в конце ключ, про который он изначально забыл), тем не менее, авторы рекомендуют придерживаться более строгой формы, допустимой во всех версиях UNIX. В противном случае в системах FreeBSD или Mac OS X вы можете столкнуться с постоянными жалобами программ на неверный порядок следования аргументов. Поверьте, мы уже это проходили!

Для иллюстрации здесь специально были подобраны простые примеры. Однако, если взглянуть на примеры использования программ командной строки, которые можно найти во Всемирной паутине (или в страницах справочного руководства man и в выводе программы с ключом --help), можно увидеть, что программы командной строки обеспечивают поразительную гибкость в терминах изменения их поведения. И это еще не все – в следующем разделе вашему вниманию будут представлены совершенно иные аспекты работы команд UNIX.

Каналы и перенаправление

При работе с командной строкой практически всегда приходится иметь дело с текстом, как при вводе команды, так и при получении результатов ее работы. Однако, кроме ввода и вывода текстовой информации, программы в системе UNIX взаимодействуют между собой и с файлами на диске посредством уровня абстракции ввода/вывода, который называется **каналами**. Как следует из названия, каналы – это механизм управления потоками информации между различными компонентами терминального устройства для организации взаимодействия с пользователями, с другими программами и файлами.

Каждая программа в системе UNIX имеет дело с тремя типами ввода-вывода: ввод, обычный вывод результатов работы и вывод информации, связанной с ошибками. В обычной ситуации программы взаимодействуют с так называемыми стандартными каналами, связанными с текстовым терминалом пользователя. Например, когда используется команда cat (сокращенно от «concatenate» – объединение[1]), чтобы вывести содержимое файла, эта команда открывает файл и выводит его содержимое в поток стандартного вывода stdout, как показано ниже, где выводится содержимое файла, представляющее собой список бакалейных товаров.

```
$ cat groceries.txt
Milk
Canned corn
Peanut butter
Can of soup
Powdered milk
```

Здесь мы просто вызываем команду cat, поэтому информация из потока stdout попадает на наш терминал. Если команда cat столкнется с ошибочной ситуацией, например, если она получит имя несуществующего файла, она выведет сообщение об ошибке в поток стандартного вывода сообщений об ошибках stderr, который также по умолчанию связан с терминалом пользователя.

[1] Имеется в виду, что команда cat может объединить вывод сразу из нескольких файлов. – *Прим. перев.*

```
$ cat foo.txt
cat: foo.txt: Нет такого файла
```

Вся прелесть уровня абстракции каналов заключается в том, что имеется возможность отступить от обычной схемы и использовать **оператор канала** (|), чтобы сообщить оболочке о необходимости *перенаправить* поток stdout одной программы в поток stdin другой. Название stdin происходит от «standard input» (стандартный ввод); stdin представляет третий механизм ввода-вывода программ. Многие программы могут получать текстовую информацию из потока стандартного ввода, в дополнение или вместо файлов, имена которых определит пользователь.

Теперь вспомним о команде head и применим ее к операции вывода списка товаров, чтобы получить только первый элемент списка.

```
$ cat groceries.txt | head -n1
Milk
```

Обратите внимание, что мы не указываем команде head имя файла, содержимое которого она должна выводить, вместо этого мы использовали оператор канала, чтобы перенаправить вывод команды cat на ввод команды head. Результат получился тот же, как если бы мы напрямую указали имя файла команде head.

```
$ head -n1 groceries.txt
Milk
```

В качестве более практичного примера рассмотрим использование утилиты grep, которая, кроме всего прочего, может использоваться, чтобы получить только те строки, которые соответствуют заданному регулярному выражению (за дополнительной информацией обращайтесь к главе 1 «Практическое введение в Python для Django»). Давайте попробуем с помощью каналов взять вывод команды grep, которая извлекает из списка (без учета регистра символов) только те элементы, которые содержат слово «can»:

```
$ grep -i "can" groceries.txt
Canned corn
Can of soup
```

и передать его команде head, чтобы получить только первый элемент.

```
$ grep -i "can" groceries.txt | head -n1
Canned corn
```

Как уже упоминалось выше, в одной команде можно использовать несколько каналов. Давайте попробуем дополнительно задействовать команду sed, родственную команде grep, чтобы заменить слово «corn» (кукуруза) на более универсальное «veggies» (овощи).

```
$ grep -i "can" groceries.txt | head -n1 | sed -e "s/corn/veggies/"
Canned veggies
```

Наконец, как уже говорилось, имеется возможность перенаправить потоки ввода и вывода, связав их с файлами с помощью операторов перенаправления < и > соответственно. Запомните, что операции с каналами выполняются в направлении слева направо, оператор > используется для направления потока stdout в файл, а оператор < – для направления содержимого файла в поток stdin. Например, в предыдущем примере, в котором производились поиск и замена, после отображения введенного на терминале наша тяжелая работа завершена (за исключением операции копирования и вставки, конечно). Давайте попробуем сразу же перенаправить получаемые результаты в файл.

```
$ grep -i "can" groceries.txt | head -n1 | sed -e "s/corn/veggies/" >
filtered.txt
```

Эта команда создаст новый файл (или перезапишет существующий, поэтому будьте очень внимательны!) с именем filtered.txt и запишет в него строку «Canned veggies» (консервированные овощи). Обратите внимание, что теперь команда ничего не выводит на терминал. Это обусловлено тем, что мы *перенаправили* поток stdout в файл, а так как информация была перенаправлена, на терминале она не отобразилась.

И последнее, следует отметить, что использование удвоенного символа перенаправления (>>) *добавляет* информацию в конец существующего файла, а не перезаписывает его. Если файл не существует, он будет создан, как и в случае с оператором >.

Переменные окружения

Командная оболочка имеет то, что называется **окружением**, или пространством имен, аналогично пространствам имен программ на языке Python или других языках программирования, где различные строки связываются с именами переменных и используются для ссылки на них при выполнении команд пользователем или даже самими командами (которые имеют доступ к окружению, откуда был произведен вызов). Текущее состояние окружения можно получить с помощью команды env, как показано ниже:

```
$ env
TERM=linux
SHELL=/bin/bash
USER=user
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games:~/bin
PWD=/home/user
EDITOR=vim
HOME=/home/user
```

Многие переменные окружения используются общими утилитами системы UNIX или самой командной оболочкой. Например, переменная EDITOR используется такими программами, как Subversion, когда возникает необходимость временно переместить пользователя в тек-

стовый редактор. Переменная TERM определяет тип терминала, что часто бывает необходимо программам для вывода информации в цвете или для интерпретации комбинаций клавиш. Переменная PWD хранит имя текущего каталога и т. д.

Как в языке Python и в других языках программирования, имеется возможность изменять значения переменных окружения с помощью оператора присваивания. В следующем примере мы для компактности сократили вывод команды env.

```
$ env
EDITOR=vim
$ EDITOR=pico
$ env
EDITOR=pico
```

В этом примере значением по умолчанию для переменной EDITOR является широко известный редактор vim. Мы изменили это значение, и теперь переменная указывает на менее мощный текстовый редактор pico. Переменные окружения, как и можно было предположить, инициализируются в начале каждого сеанса работы с командной оболочкой. Наше изменение переменной EDITOR носит временный характер, если только не изменить содержимое конфигурационных файлов командной оболочки, содержимое которых она читает в начале каждого сеанса, чтобы сделать изменения постоянными. (Чтобы узнать, как это сделать, обращайтесь к страницам справочного руководства для вашей командной оболочки.)

До настоящего момента мы использовали команду env, чтобы вывести значения переменных окружения, однако, как и в случае с функцией globals в языке Python, в этом мало проку, если вы не занимаетесь поисками чего-то определенного. Командная оболочка может автоматически определять использование переменных окружения и подставлять на их место их значения, но только когда имена переменных предваряются символом $. Для демонстрации сказанного воспользуемся командой echo, которая просто выводит свои аргументы:

```
$ env
EDITOR=vim
$ echo EDITOR
EDITOR
$ echo $EDITOR
vim
```

Как видите, при попытке вывести строку EDITOR не происходит ничего особенного – это просто строка, но когда используется аргумент $EDITOR, команда выводит значение переменной EDITOR. Другими словами, командная оболочка извлекает имя или выражение, следующее за символом $, и пытается получить значение переменной. Те из нас, кто привык использовать языки программирования, где символ $ применяется для обозначения переменных как в инструкциях присваива-

ния, так и для получения их значений, часто совершают ошибки, такие как показано ниже:

```
$ $EDITOR=pico
bash: vim=pico: Нет такого файла или каталога
$ EDITOR=pico
$ echo $EDITOR
pico
```

Последнее замечание по этой теме: значения переменных окружения не ограничиваются простыми строками, состоящими из одного слова, они могут содержать любые строки. Как видно из предыдущего примера, командная оболочка получила значение переменной $EDITOR, подставила его в строку и попыталась выполнить ее как команду. Очевидно, что такой прием не сработал, потому что в системе нет программы с именем vim=pico. Но эту особенность вполне можно использовать, чтобы сэкономить на вводе с клавиатуры, как показано в следующем примере, где большая часть строки команды помещена в переменную и используется несколько раз с разными аргументами.

```
$ FINDMILK="grep -ni milk"
$ $FINDMILK groceries.txt
1:Milk
5:Powdered milk
$ $FINDMILK todo.txt
1:Search grocery list for milk
$ $FINDMILK email_from_reader.txt
3:Also, what's up with all the groceries and milk examples?
```

Каждый раз на место нашей переменной FINDMILK подставляется ее значение, в результате команда приобретает вид, например, grep -ni milk groceries.txt. Однако этот пример достаточно надуманный – во многих случаях, подобных этому, когда возникает необходимость обеспечить возможность передачи *различных аргументов* некоторой статической команде, лучше написать небольшой сценарий командной оболочки. Например, можно было бы написать сценарий, принимающий не один, а два аргумента и позволяющий указывать не только место поиска, но и предмет поиска.

Описание подробностей создания сценариев на языке командной оболочки выходит далеко за рамки этой главы, но на странице справочного руководства man к своей командной оболочке вы найдете массу информации. Кроме того, как и в любых других случаях, огромное число замечательных руководств можно найти в Интернете.

Пути

Одна из, пожалуй, самых важных переменных окружения – это переменная PATH, которая хранит пути поиска, то есть список каталогов, просматриваемых командной оболочкой при поиске введенной команды.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:~/bin
```

Когда пользователь вводит команду, командная оболочка пытается отыскать исполняемый файл с заданным именем во всех перечисленных каталогах. Если такой файл будет найден, он запускается с указанными вами ключами и аргументами. Так, например, когда вводится команда echo, командная оболочка фактически выполняет файл /bin/echo, а когда вводится команда man, она отыскивает файл /usr/bin/man. (Различия между этими двумя каталогами bin описываются во врезке, следующей ниже.)

Имеется возможность добавлять в переменную PATH собственные пути, чтобы сэкономить на вводе с клавиатуры. В предыдущем примере пользователь добавил каталог ~/bin, а так как символ ~ является сокращенным именем домашнего каталога, пользователь тем самым получил возможность вводить в командной строке имена сценариев из своей личной библиотеки, которые благополучно будут обнаруживаться командной оболочкой.

Обратите внимание, что вы без труда можете запускать программы, находящиеся вне пути поиска, для чего достаточно просто указать абсолютный или относительный путь к выполняемому файлу. В действительности путь поиска в командной оболочке – это лишь удобный способ запускать программы.

```
$ /tmp/packagename-2.0.1/bin/program
```

Поскольку каталог /tmp/packagename-2.0.1/bin/ (вероятное местонахождение выполняемых файлов программы, которую вы пытаетесь опробовать перед окончательной установкой) наверняка не входит в список каталогов, хранящийся в переменной PATH, вам необходимо явно указать каталог, в котором командная оболочка должна отыскать выполняемый файл. В действительности вся концепция путей поиска – это всего лишь возможность сэкономить на вводе с клавиатуры, хотя и очень распространенная.

Наконец, имейте в виду, что в переменной PATH перечисляются каталоги, где находятся выполняемые файлы, а не сами выполняемые файлы – то же относится и к другим путям поиска, таким как путь поиска модулей в языке Python (см. главу 1). Путь поиска – это список контейнеров, где следует искать, а не список предметов, которые можно отыскать.

Каталоги bin в системе UNIX

В системе UNIX традиционно имеется несколько каталогов для размещения разных классов программ. Например, в каталог /usr/bin помещаются обычные двоичные файлы (такие выполняемые файлы, как утилиты, которые мы используем в примерах этого приложения), в каталог /usr/local/bin помещаются программы, устанавливаемые пользователем (в противоположность тем, что устанавливаются менеджером пакетов операционной системы). При этом большая часть выполняемых файлов, предназначенных для использования администратором системы, а не обычными пользователями, может находиться в каталогах /sbin и /usr/sbin.

Некоторые сторонние приложения, такие как реализации Java или устаревшие версии пакета программ Mozilla, помещают свои данные в собственные каталоги, такие как /opt/mozilla, а выполняемые файлы – в каталог /opt/mozilla/bin. Обычно пути поиска включают в себя подмножество системных каталогов с выполняемыми файлами, собственные каталоги сторонних программ и некоторые другие.

В заключение

К настоящему моменту вы многое уже узнали: как выполнять программы с различными ключами и аргументами, как обеспечить взаимодействие этих программ между собой и с файлами посредством каналов и операторов перенаправления и как переменные окружения и пути поиска могут сэкономить вам массу времени.

Однако типичные командные оболочки UNIX – это намного больше, чем было показано здесь; большинство из них представляют собой полноценные среды программирования, со своими особенностями, включая условные инструкции, циклы и т. д. Как только вы почувствуете, что без труда можете перемещаться по файловой системе и запускать команды, вам имеет смысл заняться более глубоким исследованием командной оболочки, используемой в вашей системе, – это позволит вам сэкономить немало времени, наравне с другими инструментами программирования.

B

Установка и запуск Django

Чтобы приступить к разработке веб-приложений на платформе Django, необходимо установить ее. Способ установки зависит от используемой операционной системы и инструментов. В простейшем случае для работы платформе Django необходимы наличие интерпретатора Python 2.5, база данных SQL и веб-сервер, встроенный в платформу Django.

Однако, чтобы развернуть платформу Django для эксплуатации в рабочем режиме, как правило, необходим набор более надежных приложений, таких как веб-сервер Apache и база данных PostgreSQL. В этом приложении вашему вниманию предлагается несколько вариантов развертывания платформы Django, включая дополнительные указания о наиболее типичных конфигурациях. Приложение делится на основные части, описывающие развертывание языка Python и самой платформы Django, веб-сервера и системы управления базами данных.

Python

Как и в случае любого другого языка программирования, при установке Python лучше выбирать наиболее свежую его версию. Платформа Django может работать с любыми версиями Python, начиная с версии 2.3, но мы рекомендуем выбирать наиболее свежую, стабильную версию. (К моменту написания этих строк таковой была версия Python 2.5.2 и приближался параллельный выход версий 2.6 и 3.0.) Описания различий между этими последними версиями Python вы найдете в главе 1 «Практическое введение в Python для Django».

Пакеты установки для всех основных платформ можно получить на официальном сайте Python по адресу *http://www.python.org/download/*. Ниже приводятся несколько указаний, касающихся определенных платформ.

Если вы не уверены, какая версия Python уже установлена в системе (если вообще установлена), введите в командной оболочке команду `python -V` (обратите внимание – это *заглавная* буква «V»). Если интерпретатор Python установлен, он сообщит номер версии и закончит работу.

```
$ python -V
Python 2.5.1
```

Mac OS X

В операционной системе Mac OS X интерпретатор Python устанавливается по умолчанию. Если вы используете версию OS X 10.5 («Leopard»), то в ее составе поставляется Python 2.5; в противном случае у вас будет стоять версия Python 2.3, и тогда мы рекомендуем вам обновить версию. Версии Python 2.4 и 2.5 можно получить непосредственно на сайте *python.org*, в виде установочных пакетов. Можно также выполнить обновление системы из портов MacPorts (*http://macports.org*). Если вы решили использовать MacPorts, вам также следует установить порт `python_select`, чтобы получить возможность назначить более новую версию Python версией, используемой системой по умолчанию.

UNIX/Linux

Большинство открытых UNIX-подобных систем, таких как Linux или семейство операционных систем BSD, также устанавливают Python по умолчанию, как часть базовой системы. Конкретная версия интерпретатора может варьироваться в широких пределах, в зависимости от разновидности операционной системы и времени ее выхода. С помощью системы управления пакетами, входящей в состав дистрибутива, проверьте, имеется ли возможность установить наиболее свежую версию Python, или посетите страницу загрузки на сайте *python.org*, где можно найти установочные пакеты, такие как RPM или архивы с исходными текстами.

Windows

По умолчанию Python отсутствует в операционной системе Windows, поэтому вам необходимо зайти на официальную страницу загрузки Python. Как вариант, можно посетить сайт книги «Core Python» (*http://corepython.com*) и щелкнуть на ссылке Download Python (Загрузить Python), которая находится на странице слева. После этого вы увидите таблицу со списком текущих версий Python для всех платформ. Выберите в списке самую последнюю стабильную версию для своей системы.

Существует также библиотека для языка Python, доступная только для Windows, известная под названием Python Extesions for Windows (она же `win32all`), позволяющая разрабатывать на языке Python приложения для платформы Windows. Пользователи, которые только начинают знакомиться с языком Python, могут попробовать отыскать пакет

интегрированной среды разработки PythonWin, даже если им не нужна тесная интеграция с операционной системой.

Обновление путей поиска

После установки Python вам может потребоваться добавить каталог с исполняемым файлом интерпретатора в путь поиска системы. Как правило, в UNIX-подобных операционных системах, таких как Linux и Mac OS X, все необходимое уже сделано заранее, если Python устанавливается в один из предопределенных каталогов, такой как `/usr/bin`, `/usr/local/bin` и т. д. Однако пользователи Windows должны выполнить этот шаг вручную, чтобы иметь возможность запускать сценарии на языке Python из командной строки. Как это сделать, рассказывается ниже:

Щелкнуть правой кнопкой мыши на ярлыке Мой Компьютер (My Computer) и выбрать в появившемся контекстном меню пункт Свойства (Properties). В открывшемся диалоге Свойства системы (System Properties) выберите вкладку Дополнительно (Advanced), как показано на рис. B.1.

На вкладке Дополнительно (Advanced) вы увидите три основных раздела (рис. B.2). Пропустите их и щелкните на кнопке Переменные среды (Environment Variables), что находится внизу окна диалога.

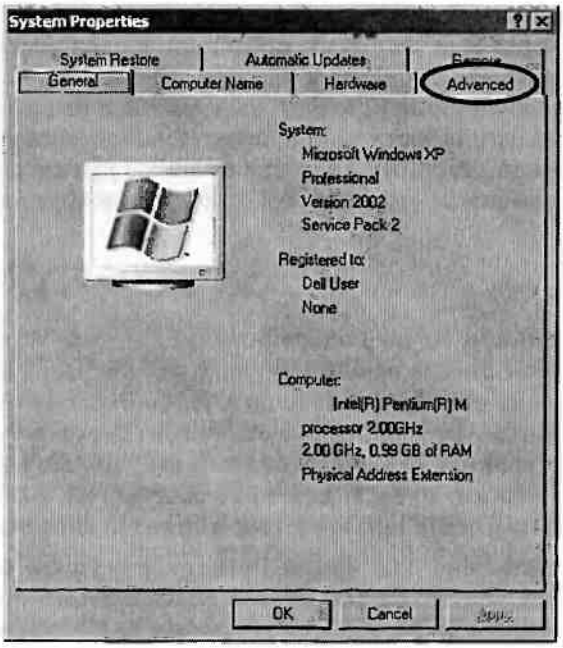

Рис. B.1. Диалог «Свойства системы» (System Properties)

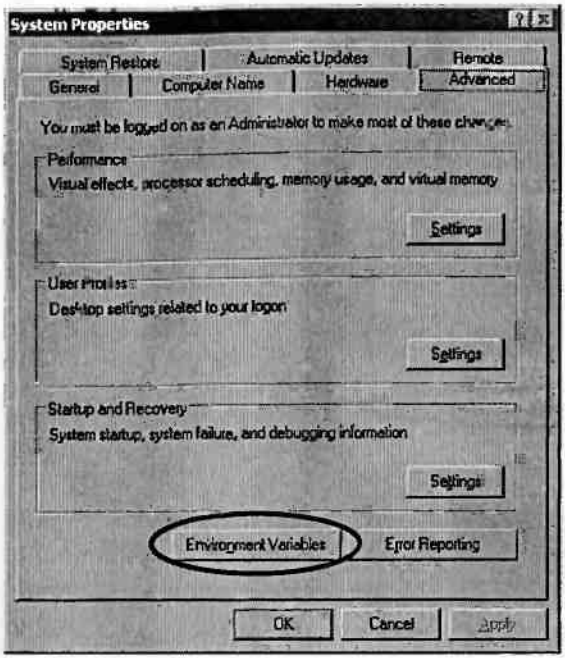

Рис. В.2. Вкладка «Дополнительно» (Advanced)

После щелчка на этой кнопке появится диалог с двумя панелями (рис. В.3), с помощью которого можно изменять значения переменных окружения. Здесь вы можете добавить/изменить/удалить переменную PATH только для себя (в панели «Переменные среды пользователя для *USER*» – «User variables for *USER*») или для всех пользователей системы (в панели «Системные переменные» – «System variables»), если вы имеете право на это.

Выберите переменную, которую требуется изменить, щелкните на кнопке Изменить (Edit) в соответствующей панели и добавьте в список строку C:\Python25, следуя формату определения переменной, как показано на рис. В.4.

Если в списке уже присутствуют какие-либо каталоги, вы можете добавить свою строку в любое место в списке, главное, чтобы все папки отделялись друг от друга точкой с запятой. Если такой переменной нет, добавьте ее – нет ничего плохого в том, что она будет содержать путь к единственному каталогу.

Щелкните на кнопке OK и откройте окно программы Командная строка (DOS/Command shell); теперь вы имеете возможность запустить интерпретатор, не указывая полный путь к нему C:\Python25\python.exe – должно быть достаточно одной команды python (смотрите ниже).

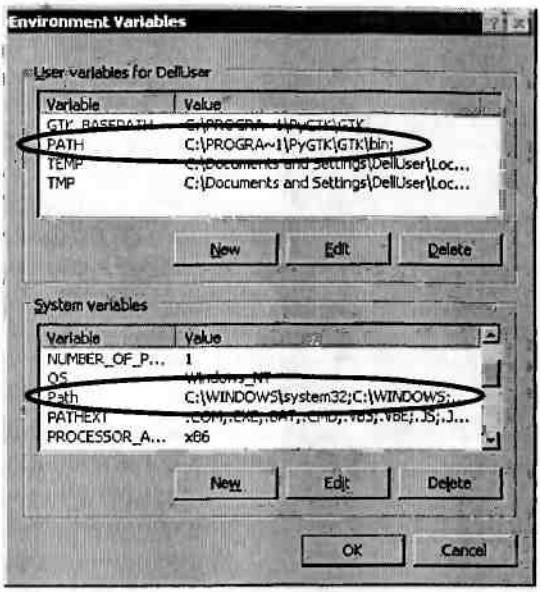

Рис. В.3. Переменные окружения

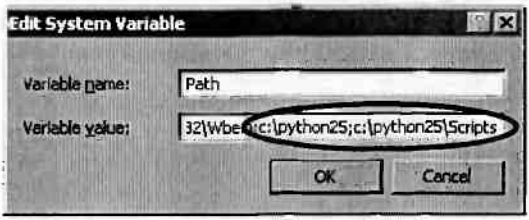

Рис. В.4. Изменение переменной PATH

Тестирование

Чтобы убедиться, что установка Python прошла гладко, просто запустите интерактивный сеанс работы с интерпретатором, выполнив главную программу. Введите команду `python` в окне терминала, при этом в некоторых случаях может потребоваться добавить номер версии, например `python2.4`. Если все работает правильно, на экране появятся результаты запуска интерпретатора, которые в текстовом контексте имеют примерно следующий вид:

```
$ python
Python 2.5.1 (r251:54863, Mar 7 2008, 04:10:12)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Или, если установка производилась в системе Windows, окно программы Командная строка (DOS/Command shell) должно выглядеть примерно так, как показано на рис. В.5.

```
MS-DOS - python
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>python
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Рис. В.5. *Результаты запуска интерпретатора Python в окне программы cmd.exe*

Три символа >>> означают, что вы находитесь в интерактивной оболочке интерпретатора, которая ожидает ввода корректного программного кода на языке Python. Выход из оболочки осуществляется нажатием комбинации клавиш Ctrl-D (в UNIX или в IDLE) или ^Z (в окне программы Командная строка (DOS/Command shell)).

Если при попытке запустить интерпретатор вы получили сообщение об ошибке, такое как «команда не найдена» или «команда 'python' не является внутренней или внешней командой», это может свидетельствовать, что вы добавили не тот каталог в переменную PATH. Проверьте ее еще раз, уточните, в какой каталог была выполнена установка, проверьте наличие в нем файла python.exe и сравните его с именем каталога, который вы добавили в переменную окружения PATH.

Наконец, для большего удобства добавьте также в путь поиска свой каталог Scripts, проделав те же самые действия, что и выше. Это позволит вам запускать инструмент администрирования Django django-admin.py так же, как сам интерпретатор Python.

Наши поздравления! Если это ваша первая попытка использовать Python и раньше вам не приходилось читать какие-либо учебные руководства или главу 1, мы предлагаем вам сделать это прямо сейчас, хотя бы бегло, и лишь затем продолжить эксперименты. Никогда не стоит пренебрегать возможностью провести некоторое тестирование инструмента, прежде чем начинать использовать его всерьез.

13 Зак. 1496

Необязательные дополнения

Наряду с установкой Python мы можем порекомендовать установить следующие инструменты (хотя это и необязательно): Easy Install и IPython.

Easy Install

Одна из самых сильных сторон Python состоит в том, что он поставляется «в комплекте с батарейками» – богатой стандартной библиотекой модулей и пакетов, которые помогут решить широкий круг задач. Если окажется так, что стандартной библиотеки будет недостаточно для вашего приложения, в вашем распоряжении богатейший выбор программного обеспечения сторонних разработчиков. Поэтому, прежде чем приступать к изобретению своего колеса, загляните в каталог пакетов Python Package Index, или «PyPI», по адресу: *http://pypi.python.org*; возможно, необходимый вам инструмент уже был кем-то создан.

По мере расширения круга необходимых вам дополнительных инструментов управление установкой Python будет становиться все более обременительным занятием. Вам придется беспокоиться о таких аспектах, как совместимость инструментов с вашей версией Python и с другим программным обеспечением, необходимо будет заниматься интеграцией новых пакетов и модулей в вашу установку Python, чтобы они могли бы импортироваться вашими приложениями, и т. д.

К счастью, имеется инструмент, способный решать все эти задачи за вас, который называется Easy Install. Получить его можно по адресу *http://peak.telecommunity.com/DevCenter/EasyInstall*. Вам достаточно загрузить и выполнить единственный файл `ez_setup.py` (обладая при этом правами администратора или с помощью команды `sudo`), после чего установка новых пакетов сможет выполняться простой командой:

```
easy_install НОВОЕ_СТОРОННЕЕ_ПРОГРАММНОЕ_ОБЕСПЕЧЕНИЕ
```

Инструмент Easy Install использует каталог «PyPI», чтобы определить наличие последней (или определенной) версии требуемого программного обеспечения, загрузить его (*и* все его зависимости) и установить его, – и все это с помощью единственной команды.

Обновление и удаление программных пакетов выполняется так же просто.

IPython

Инструмент IPython – это сторонняя альтернатива стандартному интерактивному интерпретатору, который поставляется в составе Python. Он добавляет массу удобных особенностей сверх того, что дает интерпретатор Python, включая функцию дополнения имен переменных и атрибутов по нажатию на клавишу табуляции, хронологию команд, автоматическое оформление отступов, простой доступ к строкам доку-

ментирования и сигнатурам функций и многое другое. Благодаря популярности и богатству возможностей IPython этот инструмент может заменить вам интерактивный интерпретатор по умолчанию, когда вы начнете использовать сценарий manage.py администрирования своих проектов на платформе Django. Дополнительную информацию об инструменте IPython вы найдете на сайте *http://ipython.scipy.org*.

В качестве альтернативы операции загрузки с веб-сайта, а также с целью продемонстрировать инструменты Easy Install *и* IPython; особенно, чтобы показать, насколько просто выполняется установка сторонних пакетов с помощью Easy Install, ниже приводится процесс установки IPython инструментом Easy Install в системе Linux (при использовании команды sudo для получения привилегий администратора):

```
$ sudo easy_install ipython
Password:
Searching for ipython
Reading http://pypi.python.org/simple/ipython/
Reading http://ipython.scipy.org
Reading http://ipython.scipy.org/dist
Best match: ipython 0.8.4
Downloading http://ipython.scipy.org/dist/ipython-0.8.4-py2.4.egg
Processing ipython-0.8.4-py2.4.egg
creating /usr/lib/python2.4/site-packages/ipython-0.8.4-py2.4.egg
Extracting ipython-0.8.4-py2.4.egg to /usr/lib/python2.4/site-packages
Adding ipython 0.8.4 to easy-install.pth file
Installing ipython script to /usr/bin
Installing pycolor script to /usr/bin

Installed /usr/lib/python2.4/site-packages/ipython-0.8.4-py2.4.egg
Processing dependencies for ipython
Finished processing dependencies for iPython
```

В системе Windows процесс установки выглядит практически так же — при условии, что вы обладаете правами администратора. Ниже приводится процесс установки IPython в Windows, где ранее уже была установлена последняя версия этого инструмента:

```
C:\>easy_install ipython
Searching for ipython
Best match: ipython 0.8.2
Processing ipython-0.8.2-py2.5.egg
ipython 0.8.2 is already the active version in easy-install.pth
Deleting c:\python25\Scripts\ipython.py
Installing ipython-script.py script to c:\python25\Scripts
Installing ipython.exe script to c:\python25\Scripts
Installing pycolor-script.py script to c:\python25\Scripts
Installing pycolor.exe script to c:\python25\Scripts

Using c:\python25\lib\site-packages\ipython-0.8.2-py2.5.egg
Processing dependencies for ipython
Finished processing dependencies for iPython
```

Django

Теперь, когда установлен и настроен интерпретатор языка Python, необходимо установить саму платформу Django. На момент работы над этой книгой последней версией Django была версия 1.0 и велись работы над версией 1.1. Мы рекомендуем вам использовать версию 1.0 – во-первых, потому что эта книга в значительной степени основана на версии 1.0, а во-вторых, потому что всегда желательно использовать последнюю стабильную версию. Однако для тех, кто стремится всегда быть на передовом крае, ниже приводится информация об использовании версии Django, находящейся в разработке.

Официальные выпуски

Официальные выпуски Django можно получить на веб-сайте проекта по адресу *http://www.djangoproject.com/download/* (или посредством менеджеров пакетов – узнайте, возможно ли это с помощью имеющейся у вас системы пакетов). Официальные выпуски, распространяемые через веб-сайт, поставляются в типичном для UNIX формате `.tar.gz` – системы UNIX и Mac изначально способны открывать файлы в этом формате, а вот пользователям Windows потребуется дополнительное программное обеспечение, например 7-Zip (*http://7zip.org*), инструмент командной строки LibArchive (*http://gnuwin32.sf.net/packages/libarchive.htm*) или среда «Unix-on-Windows», такая как Cygwin (*http://www.cygwin.com/*).

Версия в разработке

Чтобы получить версию Django, находящуюся в разработке и включающую в себя самые последние особенности, отсутствующие в последней стабильной версии, необходимо установить клиентскую часть системы управления версиями Subversion. Как и Python, программное обеспечение Subversion может быть установлено с помощью менеджеров пакетов (в UNIX) или портов MacPorts (в OS X); в противном случае необходимо напрямую загрузить дистрибутив и установить его – подробности смотрите на веб-сайте Subversion (*http://subversion.tigris.org/*).

Как только программное обеспечение Subversion будет установлено, вы сможете получить самую последнюю версию Django всего одной командой:

```
$ svn co http://code.djangoproject.com/svn/django/trunk django_trunk
```

Установка

После распаковывания файла `.tar.gz` (в случае официального выпуска) или извлечения исходных текстов Django из репозитория (в случае версии, находящейся в разработке) в текущем каталоге появится новый подкаталог с именем `Django-1.0` или `django_trunk`. Внутри этого

подкаталога находятся все компоненты платформы Django – не только сама платформа (пакет на языке Python в каталоге `django`), но еще документация и тесты, а также другие сценарии и дополнительная информация.

Чтобы приступить к работе с Django, необходимо выполнить одно из следующих действий:

- Добавить новый каталог в переменную окружения `PYTHONPATH`. Если вы извлекали исходные тексты из репозитория в каталог `/home/username`, например вам следует добавить в переменную `PYTHONPATH` каталог `django_trunk`, а не подкаталог `django`.
- Переместить, скопировать или добавить символическую ссылку на каталог `django` в каталог Python `site-packages`.
- Войти в новый каталог и выполнить команду `python setup.py install` (обладая при этом правами администратора), в результате платформа Django будет автоматически установлена в нужное место.

Более подробно механизм путей поиска Python описывается в главе 1. Заметим, что если вы планируете использовать Subversion, чтобы постоянно иметь самую свежую версию, находящуюся в разработке, то предпочтительнее использовать первые два варианта (причем во втором варианте желательно следовать только рекомендации о символической ссылке). Вариант с использованием команды `setup.py install` при использовании версии из репозитория также будет работать, но вам постоянно придется выполнять эту команду при каждом обновлении версии с помощью Subversion, поэтому не рекомендуется использовать этот вариант.

Тестирование

Чтобы убедиться, что в переменной `PYTHONPATH` был указан правильный каталог с платформой Django, запустите интерактивный сеанс интерпретатора Python и попробуйте выполнить инструкцию `import django`. Если сообщений об ошибках не появилось, значит, все было сделано правильно! Если вы получили сообщение, такое как `ImportError: No module named django` (не найден модуль с именем django), дважды проверьте еще раз все свои действия или попробуйте реализовать другой вариант из тех, что были представлены выше.

Веб-сервер

Установка Python и Django – это первый большой шаг к фактическому использованию Django. Следующим важнейшим аспектом является веб-сервер, который решает задачу доставки динамических страниц HTML броузерам. Кроме того, веб-сервер отвечает за обслуживание статических файлов содержимого, таких как изображения и файлы CSS, не говоря уже о решении других задач системного уровня, возни-

кающих при разработке веб-приложений (балансировка нагрузки, проксирование и т. д.).

Встроенный сервер: не для работы в нормальном режиме эксплуатации

Простейшим веб-сервером, который можно использовать для работы с платформой Django, является встроенный сервер разработки, который основан на встроенном модуле `BaseHttpServer` языка Python (хороший пример использования стандартной библиотеки языка Python). Вы уже имели возможность видеть этот сервер в действии в главе 2 «Django для нетерпеливых: создание блога», где он был задействован с помощью сценария `manage.py` для опробования простого веб-приложения. Это именно та область, где наиболее ярко проявляются преимущества сервера разработки: проверка новых идей и быстрая разработка основы будущего приложения. Он также прекрасно подходит для отладки, так как выполняется на переднем плане в окне терминала, благодаря чему можно наблюдать все сообщения, выводимые инструкциями `print`.

Однако создание высокопроизводительных, надежных и защищенных веб-серверов – это не самая простая задача, и потому команда разработчиков Django поступила мудро, отказавшись от идеи изобретения своего колеса. Встроенный сервер разработки не предназначен для развертывания в качестве веб-сервера в общедоступном Интернете и абсолютно ни на что не пригоден, кроме решения задач тестирования и разработки простых веб-приложений. Документация Django грозит (в шутку) «отзывом лицензии на платформу Django», если вы будете использовать сервер разработки в режиме нормальной эксплуатации, и это предупреждение вполне обосновано.

Наконец, хотя сервер разработки обладает возможностью обслуживать статические файлы (смотрите официальную документацию или информацию на сайте *withdjango.com*), тем не менее, мы настоятельно рекомендуем, чтобы вы позаботились о настройке одного из следующих вариантов веб-серверов. Ведь нет никаких веских причин откладывать неизбежное, а кроме того, вы получите возможность вести разработку в условиях, максимально приближенных к эксплуатационным, – любые ошибки и проблемы лучше обнаруживать раньше, чем позже.

Стандартный подход: Apache и mod_python

Веб-сервер Apache и его модуль `mod_python` уже долгое время остаются предпочтительным способом развертывания сайтов на платформе Django. Именно эту комбинацию использовал Лоуренс (Lawrence) и его команда, создатели платформы Django, использовали в качестве основы своих высоко нагруженных новостных веб-сайтов; и она и по сей день

представляет собой хорошо протестированный и хорошо документированный вариант развертывания.

Если у вас имеются требования, в которые не вписывается использование модуля mod_python, например при использовании услуг хостинга, предоставляемых сторонней компанией, или когда эксплуатируется веб-сервер, отличный от Apache, смотрите следующие разделы, где описываются альтернативные варианты, основанные на применении WSGI и Flup. Однако, если сервер (или экземпляр виртуального сервера) находится полностью под вашим контролем или поддерживает модуль mod_python, то это самый надежный путь.

Для работы с платформой Django вам необходимо установить Apache версии 2.2 или 2.0 (иногда в менеджерах пакетов он называется apache2, чтобы обозначить отличие от Apache 1.3 – устаревшей версии веб-сервера) и модуль mod_python версии 3.0 или выше. Как и в случае с Python, фактическая процедура установки веб-сервера может существенно отличаться в зависимости от платформы; так, пользователи Mac версии 10.4 и ниже должны использовать порты MacPorts (потому что Apache 2 имеется только в версии «Leopard»). Пользователи Windows могут найти скомпилированные версии веб-сервера и модуля mod_python на сайтах *http://httpd.apache.org* и *http://www.modpython.org*, соответственно.

Что касается настройки после установки, то существует две основные проблемы, которые необходимо решить при развертывании Django с модулем mod_python: как будет подключаться к веб-серверу сама платформа Django, и где будут находиться статические файлы.

Подключение Django к веб-серверу Apache

Первое, что необходимо определить, – какую часть пространства адресов URL будет обслуживать платформа Django – весь сайт, например, *www.example.com*, или только подраздел или подразделы, такие как *www.example.com/foo/* (где раздел *www.example.com/* или *www.example.com/bar/* могут обслуживаться, например, с помощью PHP или содержать статические страницы HTML). Кроме того, имеется возможность организовать обслуживание нескольких проектов на платформе Django, каждый в своем подразделе сайта.

Чтобы подключить проект на платформе Django, необходимо добавить в конфигурационный файл веб-сервера Apache следующий фрагмент, который обычно должен располагаться внутри блока <VirtualHost> или, если вы не используете виртуальные хосты, внутри основного раздела файла apache2.conf (в некоторых системах – httpd.conf).

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
```

```
    PythonDebug On
</Location>
```

Предыдущий блок <Location> подключает проект mysite на платформе Django к домену верхнего уровня, управляемого файлом конфигурации, в котором он находится. Чтобы настроить платформу Django на работу с определенным сегментом адресного пространства URL, достаточно просто изменить определение <Location>, как показано ниже:

```
<Location "/foo/">
```

Кроме того, здесь возможны несколько вариантов. Первый вариант, когда путь к проекту на платформе Django не включен в системную переменную PYTHONPATH и его необходимо добавить, когда Apache будет загружать модули Python. Для этого нужно просто вставить дополнительную директиву PythonPath. Если предполагать, что проект mysite (каталог, содержащий файл settings.py, корневой файл URLconf и подкаталоги с приложениями) находится в каталоге /home/user/django-stuff/mysite/, то вам следует добавить вмещающий его каталог в путь поиска модулей Python, как показано ниже:

```
<Location "/">
    SetHandler python-program
    PythonPath "['/home/user/django-stuff/'] + sys.path"
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

Если в одном и том же домене имеется несколько проектов на языке Python, то можно просто определить несколько блоков <Location>, не забывая при этом сообщить, что следует использовать отдельные экземпляры модуля mod_python в памяти; в противном случае вы можете столкнуться с неожиданным поведением. Сделать это можно посредством определения различных (произвольных) имен обработчиков Python с помощью директивы PythonInterpreter.

```
<Location "/foo/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE foosite.settings
    PythonInterpreter foosite
    PythonDebug On
</Location>

<Location "/bar/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE barsite.settings
    PythonInterpreter barsite
    PythonDebug On
</Location>
```

Как видите, имеется некоторая вариативность в том, как можно использовать модуль mod_python для обслуживания программного кода платформы Django. Дополнительную информацию об этих и других директивах Python* веб-сервера Apache можно найти в документации к модулю mod_python по адресу *http://modpython.org*.

Оформление «разрыва» адресного пространства для статических файлов

Мы благополучно настроили обслуживание веб-приложений на платформе Django веб-сервером Apache, но кое-что мы все-таки упустили из виду: наши изображения и файлы JavaScript/CSS (а также, возможно, видеофайлы, документы PDF и что-нибудь еще, что должно обслуживаться нашим веб-сайтом). Как правило, бывает необходимо размещать эти файлы в адресном пространстве URL веб-приложения, то есть, если приложение находится в сегменте адресов /foo/, то изображения и таблицы стилей могли бы находиться по адресу /foo/media/. Мы только что подключили программный код на языке Python к сегменту адресов /foo/ и теперь подошли к необходимости обеспечить доступ к статическим файлам.

Это достаточно простая задача: нам достаточно сообщить веб-серверу Apache, что он должен отключить использование модуля mod_python для заданного местоположения, используя еще один блок <Location> после блока с описанием проекта на платформе Django.

```
<Location "/foo/media/">
    SetHandler none
</Location>
```

При таких настройках запрос к адресу, например /foo/users/ будет направлен на обработку программному коду Django, но запрос на получение файла /foo/media/images/userpic.gif приведет к выполнению поиска в корневом каталоге документов Apache (определенном где-то в другом месте в файле конфигурации или в блоке определения виртуального хоста). Как и следовало ожидать, имеется возможность определить несколько таких «разрывов» в адресном пространстве, обслуживаемом модулем mod_python, благодаря чему можно иметь несколько каталогов для хранения изображений, CSS и JavaScript, например:

```
<LocationMatch "/foo/(images|css|js)/">
    SetHandler none
</LocationMatch>
```

В отличие от обычного блока <Location>, блок <LocationMatch> позволяет использовать регулярные выражения, поэтому мы просто использовали здесь регулярное выражение, позволяющее отключить специализированный обработчик от обслуживания любого из трех указанных каталогов. Этот блок обладает функциональностью, аналогичной блоку <Location>, но выглядит более аккуратно. Нет никаких причин от-

14 Зак. 1496

казываться от принципа «не повторяйся», который мы использовали в программном коде на языке Python, – вы можете с тем же успехом применять его для настройки системы!

Гибкая альтернатива: WSGI

WSGI (Web Server Gateway Interface – шлюзовой интерфейс веб-сервера) и mod_wsgi – это восходящие звезды технологии веб-хостинга, позволяющие использовать язык программирования Python. Платформа Django обладает достаточно полной поддержкой WSGI и все большее число программистов, использующих Django (и вообще веб-программистов, использующих язык Python), предпочитают использовать ее, а не mod_python. WSGI – это гибкий протокол, позволяющий связать программный код на языке Python и любой совместимый веб-сервер – не только Apache, но и такие альтернативы, как lighttpd (*http://lighttpd.net/*), Nginx (*http://nginx.net*), CherryPy (*http://cherrypy.org*) и даже Microsoft IIS.

Несмотря на то, что технология WSGI считается достаточно новой, тем не менее, она может работать со всеми упомянутыми выше веб-серверами и протестирована на совместимость с большим числом веб-платформ на языке Python, включая Django и такие популярные веб-приложения на языке Python, как механизм wiki MoinMoin и менеджер программных проектов Trac.

Главными достоинствами модуля mod_wsgi (помимо совместимости с широким спектром веб-серверов) являются низкое потребление памяти и высокая производительность по сравнению с модулем mod_python; единый стандартный интерфейс для всех приложений WSGI, включая приложения не на платформе Django; и возможность работы в режиме демона, что легко позволяет обеспечить «принадлежность» процесса WSGI определенному пользователю в системе.

Наиболее важным недостатком модуля mod_wsgi на момент написания этих строк была его ограниченная распространенность в различных дистрибутивах, поэтому есть вероятность, что вам придется компилировать и устанавливать его самостоятельно. Однако такое положение дел скорее всего изменится в самом ближайшем будущем, а пользователи Windows уже сейчас могут использовать неофициальные сборки mod_wsgi для Windows, доступные на основном веб-сайте проекта mod_wsgi.

Если для вашей операционной системы еще не существует скомпилированной версии, вы можете загрузить исходные тексты с веб-сайта проекта mod_wsgi, *http://modwsgi.org/*, и выполнить прилагаемые инструкции по установке. После установки будет совсем несложно запустить этот модуль в работу, а в некоторых случаях даже немного проще, чем модуль mod_python. Сначала необходимо настроить веб-сервер Apache на использование этого модуля. Для версии Apache 2 следует поместить в файл httpd.conf следующую строку:

```
LoadModule wsgi_module /usr/lib/apache2/modules/mod_wsgi.so
```

и блок:

```
Alias /media/ "/var/django/projects/myproject/media"
<Directory /var/django/projects/myproject/>
    Order deny,allow
    Allow from all
</Directory>
WSGIScriptAlias / /var/django/projects/myproject/mod.wsgi
```

Наконец, необходимо создать сценарий mod.wsgi, упомянутый в последней строке предыдущего блока:

```
import os, sys
sys.path.append('/var/django/projects')
os.environ['DJANGO_SETTINGS_MODULE'] = 'myproject.settings'

import django.core.handlers.wsgi

application = django.core.handlers.wsgi.WSGIHandler()
```

В предыдущем фрагменте необходимо /var/django/projects заменить на путь к каталогу, где находится ваш проект или проекты на платформе Django. Следует заметить, что функции sys.path.append всегда следует передавать путь к каталогу, *вмещающему* каталог проекта, то есть /var/django/projects, а не /var/django/projects/myproject.

Другой подход: flup и FastCGI

Последний потенциальный метод развертывания, который мы рассмотрим, – это модуль flup на языке Python, который не только является альтернативой использованию WSGI (первичная цель создания модуля flup), но и поддерживает похожий протокол, известный под названием FastCGI (иногда это название сокращают до аббревиатуры FCGI). Протокол FastCGI, как и WSGI, предназначен для организации взаимодействий между программным кодом приложения и веб-сервером и обладает теми же преимуществами: более высокая производительность, благодаря тому, что он выполняется в виде отдельного процесса, и более высокая защищенность, вследствие того, что работает под учетной записью пользователя, отличной от учетной записи, под которой выполняется веб-сервер.

В настоящее время FastCGI более широко поддерживается поставщиками услуг хостинга, чем WSGI, отчасти потому, что эта технология поддерживает несколько языков программирования, а отчасти потому, что она существует более продолжительное время. Таким образом, если в вашем окружении не используется веб-сервер Apache и отсутствует поддержка WSGI, то технология FastCGI представляет собой отличную альтернативу.

В официальной документации Django (смотрите прямую ссылку на сайте *withdjango.com*) имеется превосходное учебное руководство по на-

стройке FastCGI, которое мы не будем воспроизводить здесь. Как упоминается в этом документе, модуль `flup` предоставляет поддержку пары дополнительных протоколов, работающих поверх WSGI и FastCGI, – SCGI и AJP, которые могут оказаться необходимым для вас, если ваши требования к развертыванию несколько отличаются от обычных.

База данных SQL

Наконец мы пришли к последней части мозаики: у вас уже установлены Python и Django и вы подключили их к своему веб-серверу (Apache, lighthttp или какому-нибудь другому) для обслуживания динамических запросов. Однако у вас все еще отсутствует хранилище для ваших данных, роль которого играет база данных SQL. Существует масса различных баз данных, которые можно использовать для нужд Django, и каждая из них требует наличия как самого программного обеспечения, реализующего базу данных, так и библиотеки для языка Python, которая обеспечит взаимодействие с ней.

В дополнение к следующим ниже замечаниям официальная документация Django содержит конкретные замечания и описания ошибок, связанных с различными платформами (смотрите прямую ссылку на сайте *withdjango.com*). Обязательно ознакомьтесь с ними, если у вас появятся вопросы или проблемы, связанные с выбранной вами базой данных.

SQLite

Название SQLite говорит само за себя – это «легковесная» реализация базы данных SQL. В отличие от PostgreSQL, MySQL и многих коммерческих баз данных, таких как Oracle или MS SQL, база данных SQLite не является самостоятельным сервером; это просто библиотека, реализующая интерфейс доступа к дисковым файлам базы данных. Как и другие «легковесные» реализации типично сложных служб, SQLite имеет свои плюсы (простота в настройке и использовании, низкая нагрузка на сервер) и минусы (недостаточная функциональность, невысокая производительность при работе с большими объемами данных).

Вследствие всего этого SQLite прекрасно подходит для быстрого запуска процесса разработки (как вы уже видели в главе 2) или для маленьких веб-сайтов, где нет необходимости устанавливать полноценный сервер баз данных. Однако, после этапа начального обучения, а также для более серьезных случаев развертывания вам необходимо будет приобрести нечто более существенное.

Чтобы иметь возможность взаимодействовать с базой данных SQLite, вам потребуется библиотека для языка Python. Если вы используете Python 2.5, в вашем распоряжении имеется встроенный модуль `sqlite3`. Если вы пользуетесь Python версии 2.4 или ниже, обратитесь на сайт

http://www.initd.org (или, как обычно, к менеджеру пакетов), чтобы получить модуль sqlite3.

В отличие от серверов баз данных, описываемых ниже, SQLite не требует явного создания базы данных или механизма управления пользователями. Вместо этого просто выбирается каталог в файловой системе для базы данных (один из тех, что доступен веб-серверу для чтения и записи) и его имя записывается в виде параметра DATABASE_NAME в файле settings.py. После этого в указанном каталоге будет создан файл базы данных при обычном использовании утилиты manage.py syncdb.

Командная оболочка SQL для базы данных SQLite реализована в виде программы sqlite3 (иногда просто SQLite, особенно, если установлена только версия SQLite 2.x), которой при запуске необходимо передавать имя файла базы данных, например sqlite3 /opt/databases/myproject.db.

PostgreSQL

PostgreSQL (часто это название сокращают до «Postgres») – это полноценный сервер баз данных, обладающий широким спектром функциональных возможностей и давней историей развития, который является одним из ведущих приложений баз данных, распространяемых с открытыми исходными текстами. Эта база данных рекомендуется к использованию основными разработчиками Django, которые дают весьма высокие оценки ее качеству. К настоящему времени база данных Postgres достигла 8-й версии (хотя версия 7 по-прежнему поддерживается платформой Django) и доступна для всех основных платформ, хотя она и не так распространена, как MySQL.

Официальный сайт Postgres, *http://www.postgresql.org*, представляет собой лучшее место, откуда можно загрузить ее (если вы – пользователь Windows или в вашем менеджере пакетов эта база данных отсутствует). На момент написания этих строк загрузить скомпилированные версии можно было на странице *http://www.postgresql.org/ftp/binary/*. Отыщите в списке последнюю версию для вашей платформы и загрузите ее.

Чтобы иметь возможность взаимодействовать с базой данных Postgres из программного кода на языке Python, вам потребуется библиотека psycopg, желательно версии 2, которая иногда называется psycopg2. Загрузить библиотеку psycopg можно с сайта *http://initd.org/pub/software/psycopg/* или установить с помощью вашего менеджера пакетов. Одно маленькое примечание: имейте в виду, что в Django имеется два механизма баз данных, по одному для каждой версии библиотеки psycopg. Убедитесь, что задействовали правильный!

Создание баз данных и пользователей в Postgres выполняется достаточно просто – в состав Postgres по умолчанию входят отдельные утилиты командной строки, такие как createuser и createdb, названия ко-

торых говорят сами за себя. В зависимости от типа системы и метода установки Postgres, возможно, что суперпользователь уровня базы данных уже был создан без вашего участия – иногда эта роль возлагается на системного пользователя postgres, иногда это может быть ваша учетная запись. За дополнительной информацией по этому вопросу обращайтесь к соответствующей документации – либо на официальный веб-сайт Postgres, либо на сайт вашей операционной системы (если вы установили Postgres с помощью менеджера пакетов).

Как только вы узнаете имя суперпользователя и пароль, вы сможете выполнить что-нибудь похожее на следующий пример и настроить базу данных и добавить пользователя для вашего проекта на платформе Django:

```
$ createuser -P django_user
Enter password for new role:
Enter it again:
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
Password:
$ createdb -U django_user django_db
Password:
```

В этом примере создается новая учетная запись django_user базы данных, с правом создания баз данных, которая затем используется для создания новой базы данных django_db. Выполнив эти действия и указав имя пользователя и базы данных в файле settings.py (вместе с паролем, присвоенным пользователю), вы сможете использовать утилиту manage.py для создания и обновления таблиц в базе данных.

Обратите на последнюю строку Password: в примере – это Postgres предлагает ввести пароль системного пользователя, который играет роль суперпользователя базы данных. Иначе мы могли бы использовать утилиту createuser с ключом -U, чтобы указать имя учетной записи суперпользователя, – примерно так, как это было сделано при вызове утилиты cratedb.

Командная оболочка SQL для PostgreSQL реализована в виде программы с именем psql и поддерживает большинство тех же ключей, что поддерживаются утилитами createuser и createdb, такие как -U для выбора имени пользователя.

MySQL

Еще одним господствующим сервером баз данных, распространяемым с открытыми исходными текстами, является MySQL. Текущей является версия 5 (хотя версия 4 также поддерживается). В MySQL отсутствуют некоторые функциональные возможности, имеющиеся в Postgres, но этот сервер баз данных получил широкое распространение

благодаря тесной интеграции с другим распространенным языком веб-программирования – PHP.

В отличие от некоторых других серверов баз данных, MySQL включает в себя два внутренних механизма баз данных, отличающихся набором возможностей: первый механизм – это MyISAM, в котором отсутствует поддержка транзакций и внешних ключей, но имеется поддержка полнотекстового поиска. Другой механизм – InnoDB, более новый и обладающий более широкими возможностями, но в настоящее время он не поддерживает полнотекстовый поиск. Существуют и другие механизмы, но эти два используются наиболее часто.

Если вы пользователь Windows или ваш менеджер пакетов не обеспечивает возможность установить последнюю версию MySQL, то вам следует обратиться на официальный веб-сайт *http://www.mysql.com*, где вы найдете скомпилированные пакеты для большинства платформ. Для организации взаимодействий с MySQL в платформе Django отдается предпочтение библиотеке `MySQLdb`, официальный сайт которой находится по адресу *http://www.sourceforge.net/projects/mysql-python*. Вам необходимо загрузить версию 1.2.1p2 или более новую. Обратите особое внимание на номер версии – некоторые старые дистрибутивы Linux включают в себя версию 1.2.1c2, которая слишком старая и несовместима с платформой Django.

Создание баз данных в MySQL обычно выполняется с помощью многоцелевого инструмента администрирования `mysqladmin`. Как и в случае с Postgres, прежде чем вы сможете создать новую учетную запись для проекта на платформе Django, вам необходимо выяснить имя суперпользователя базы данных и пароль. Обычно суперпользователь называется `root` и чаще всего изначально не имеет пароля (что следует исправить, как только появится такая возможность). Таким образом, создание базы данных выглядит примерно так, как показано ниже:

```
$ mysqladmin -u root create django_db
```

В отличие от Postgres, управление пользователями в MySQL выполняется исключительно на уровне базы данных, поэтому, чтобы создать пользователя `django_user` базы данных, нам необходимо сразу же воспользоваться командной оболочкой SQL, которая называется `mysql`.

```
$ mysql -u root
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
(Перевод:
Выполняется чтение информации, которая будет использоваться
для автодополнения имен таблиц и столбцов
Вы можете отключить эту особенность с помощью ключа -A,
чтобы запуск выполнялся быстрее)

Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 6
```

```
Server version: 5.0.51a-6 (Debian)
(Перевод:
Добро пожаловать в программу мониторинга MySQL. Команды должны
заканчиваться ; или \q.
Ваш числовой идентификатор соединения 6
Версия сервера: 5.0.51a-6 (Debian))

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
(Перевод:
Введите 'help;' или '\h' чтобы получить справку. Введите '\c',
чтобы очистить буфер.)

mysql> GRANT ALL PRIVILEGES ON django_db.* TO 'django_user'@'localhost'
IDENTIFIED BY 'django_pass';
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Вот и все! Теперь можно изменить настройки в файле `settings.py` и приступить к использованию команд `manage.py`, имеющих отношение к базе данных.

Oracle

Последняя база данных, которая в настоящее время поддерживается платформой Django, – это коммерческое предложение компании Oracle. Если вы еще плохо знакомы с разработкой приложений баз данных, вы наверняка не будете рассматривать эту базу данных в качестве возможного варианта, потому что Oracle не так широко используется поставщиками услуг хостинга и редко бывает доступна с помощью менеджеров пакетов в Linux. Однако существуют бесплатные версии Oracle, и платформа Django может взаимодействовать с версией Oracle 9*i* или выше.

Библиотека для организации взаимодействий с базой данных Oracle из программного кода на языке Python называется `cx-oracle` и доступна по адресу: *http://cx-oracle.sourceforge.net/*.

Прочие базы данных

Поддержка баз данных платформой Django непрерывно продолжает расширяться. Лучшим источником самой свежей информации являются официальный веб-сайт и почтовая рассылка для пользователей Django. Двумя базами данных, которые в настоящее время не поддерживаются напрямую платформой Django, являются Microsoft SQL Server и IBM DB2. Независимый проект, обеспечивающий поддержку MS SQL в Django, находится на сайте Google Code: *http://code.google.com/p/django-mssql/*. Там же на Google Code имеется проект Python-DB2 поддержки базы данных IBM DB2: *http://code.google.com/p/ibm-db/*, хотя на момент написания этих строк встроить такую поддержку в Django было не так-то просто.

В заключение

Мы надеемся, что к настоящему моменту вы получили работающее окружение Django, включающее все его составные части – Python, Django, веб-сервер и базу данных. Если вы столкнулись с какими-либо неприятностями, вспомните советы, которые мы давали в главе 2, – вернитесь назад и попробуйте выполнить все шаги еще раз, при этом особое внимание обратите на то, чтобы не пропустить какие-либо шаги в документации, инструкциям которой вы следуете.

Наконец, в Django имеется собственная документация по установке (охватывающая установку всех компонентов, а не только библиотек для языка Python), которая является превосходным источником сведений. Ищите прямую ссылку на сайте *withdjango.com*.

C

Инструменты разработки для платформы Django

Веб-разработка является разработкой программного обеспечения, и далеко не в минимальном масштабе. Если ваш сайт принимает ввод пользователя в том или ином виде и выполняет с ним некоторые действия, – это уже веб-приложение, а его разработка напоминает – или должна напоминать – разработку любого другого приложения.

Если раньше вы занимались разработкой обычного программного обеспечения, для вас наверняка будет очевидной необходимость использования таких инструментов и приемов разработчика, как инструменты управления версиями, отладчики и мощные среды разработки. Вам достаточно будет просто бегло просмотреть это приложение, только чтобы убедиться, что вы ничего не забыли, что могло бы облегчить вашу жизнь.

Если раньше вы занимались дизайном, то некоторые сведения, представленные здесь, могут оказаться новыми для вас. Но не пугайтесь, мы гарантируем, что любые усилия, затраченные на изучение и использование этих инструментов, многократно окупятся увеличением производительности, повышением гибкости и душевным спокойствием.

Управление версиями

Если вы занимаетесь разработкой программного обеспечения любого вида и не пользуетесь системой управления версиями, значит, вы многое упустили. Системы управления версиями позволяют сохранить полную хронологию изменений проекта, позволяя вернуться к программному коду, существовавшему ранее в тот или иной момент времени (например, за час до того как было внесено, казалось бы, невин-

ное изменение, нарушившее нечто, что вы смогли заметить только через неделю).

Старые системы управления версиями, такие как SCCS (Source Code Control System – система управления исходными текстами) и RCS (Revision Control System – система управления изменениями), не отличались большой сложностью, они обеспечивали сохранение первоначальных версий файлов и небольших изменений между версиями файла или напротив – сохраняли последнюю версию файла и так называемые обратные изменения. Одним из недостатков таких систем было то, что управляемые файлы располагались и редактировались на одном и том же сервере.

По мере развития разработки, особенно в сообществе открытого программного обеспечения, стала совершенно очевидной непригодность существующих систем управления версиями для распределенной между отдельными группами программистов разработки программного обеспечения. Это привело к появлению современных систем управления версиями, таких как CVS (Concurrent Versions System – система управления параллельными версиями) – улучшенная распределенная версия RCS и появившийся позже проект Subversion, который позиционировался как «улучшенная версия CVS» и предназначался для ее замены.

Ствол и ветви

Обычно системы управления версиями построены по аналогии с деревом, оригинальная, или основная линия разработки называется **стволом**. Ствол может копироваться, причем эти копии становятся отдельными, самостоятельными версиями, которые, как правило, продолжают развиваться в своем направлении. Копии ствола называются **ветвями**, а решение о том, какая работа в каких ветвях выполняется, изменяется от проекта к проекту.

Одна из методологий заключается в том, чтобы вести разработку, которая может нарушить обратную совместимость или привести к нестабильности, в основном стволе, а для представления выпусков программного обеспечения создавать ветви, куда после ответвления от ствола вносятся только исправления ошибок. Другой, «противоположный», подход заключается в том, чтобы сохранять стабильность основного ствола, а все новые особенности разрабатывать в ветвях. Этот подход обеспечивает удобную возможность одновременной разработки более одной существенной особенности.

При разработке самой платформы Django используется третий подход, занимающий промежуточное положение между этими двумя: в нем создаются ветви для выпусков *и* разработки новых особенностей, а основной ствол находится посередине – он не является ни совершенно стабильным, ни совершенно нестабильным. Для изменений, оказывающих существенное влияние на стабильность платформы, создаются

отдельные ветви, а изменения, имеющие менее разрушительный характер, вносятся непосредственно в основной ствол.

Слияние

Наличие ветвей полезно для создания копий программного кода, но такая возможность не имела бы смысла, если бы не было возможности объединять изменения с основным стволом! В этом месте аналогия с деревом несколько нарушается: еще одна базовая концепция в управлении исходными текстами заключается в **слиянии** изменений в двух ветвях.

Например, недавно в архитектуре форм платформы Django были выполнены существенные перестройки, а так как эта перестройка была связана с целым рядом изменений, она была реализована в виде отдельной ветви. Работа продолжалась в течение некоторого времени, и в результате получилась значительно улучшенная версия платформы. Однако в течение этого периода в ствол также вносились различные изменения, то есть с момента отделения ветви «newforms» изменения появились не только в этой ветви, но и в основном стволе.

Существует масса теоретических обоснований, как объединить нетривиальные наборы изменений, подобные данным, но, кроме того, инструменты управления версиями обеспечивают команды, позволяющие объединять такие изменения и применять их к той или другой ветви. В случае с Django ответственные разработчики запустили пару команд, чтобы дополнить ствол изменениями из ветви «newforms», вручную откорректировали несколько фрагментов, где программа управления версиями оказалась не в состоянии объединить изменения, и все.

Теперь, когда вы получили общее представление о том, что представляет собой механизм управления версиями, рассмотрим две основные современные парадигмы управления версиями, включая краткий обзор одной-двух систем управления версиями для каждой из них.

Централизованное управление версиями

Самыми известными открытыми централизованными системами управления версиями являются CVS и Subversion, причем последняя неуклонно вытесняет первую. Из более старых систем вам могут встретиться RCS и SCCS. Из коммерческих альтернатив в этой категории можно назвать Perforce, IBM Rational ClearCase и Microsoft Visual Studio Team System.

Subversion

Система управления версиями Subversion (*http://subversion.tigris.org/*) является наиболее близкой к промышленному стандарту систем управления версиями. Она распространяется с открытыми исходными

текстами, доступна для большинства платформ, имеет высокую производительность и стабильность и хорошо зарекомендовала себя на практике.

Система Subversion реализует централизованную модель с одним главным репозиторием, к которому подключаются все пользователи. Вы можете работать с программным кодом, то есть редактировать файлы, полученные из репозитория, не поддерживая постоянное сетевое соединение, но, чтобы записать свои изменения в репозиторий или получить обновления из него, необходимо выполнить подключение. Система Subversion определяет, какие изменения были внесены вами с момента получения последнего обновления с центрального сервера, но она не дает возможности выполнить запись набора изменений без подключения к серверу.

Subversion – это система управления версиями, которая в настоящее время используется для организации главного репозитория Django по адресу *http://code.djangoproject.com*, а также множества проектов, имеющих отношение к Django, на сайте Google Code и в других местах. Кроме того, Subversion прекрасно интегрируется с Trac, инструментом управления проектами и отслеживания ошибок (подробнее о Trac рассказывается ниже).

Децентрализованное управление версиями

Децентрализованное управление версиями – это будущее. Децентрализованная система обладает теми же возможностями, что и централизованная, но дополнительной мощной особенностью таких систем является то обстоятельство, что после извлечения проекта из репозитория он сам превращается в полноценный репозиторий. При использовании распределенной схемы управления версиями вам не требуется подключаться к центральному серверу, чтобы записать изменения. Вы просто записываете их в свой локальный репозиторий. Позднее, когда кто-то другой пожелает получить изменения, они могут быть перемещены по сети.

Из открытых децентрализованных систем управления версиями можно назвать Git, Mercurial, Darcs, Bazaar, SVK и Monotone. Из коммерческих систем – BitKeeper и TeamWare.

Mercurial

Mercurial (*http://www.selenic.com/mercurial/*) – это одна из наиболее популярных распределенных систем управления версиями на сегодняшний день. Система Mercurial в основном написана на языке Python, с небольшими фрагментами, где важна высокая скорость выполнения, написанными на языке C. Разработчики системы Mercurial очень серьезно относятся к производительности, благодаря чему она была взята на вооружение такими крупными проектами, как веб-броузер Mozilla и операционная система OpenSolaris компании Sun.

Git

Одним из конкурентов систем Mercurial и Bazaar в борьбе за популярность среди открытых распределенных систем управления версиями является система Git (*http://git.or.cz*). Она написана на языке С Линусом Торвальдсом (Linus Torvalds) и другими разработчиками ядра Linux. Система Git написана в духе инструментов UNIX и изначально была создана для удовлетворения чрезвычайно сложных потребностей проекта разработки ядра Linux, где используется и по сей день. Кроме того, система Git используется проектом Ruby on Rails и многими другими проектами в сфере Rails, проектами WINE и X.org (графическая система Linux), создателями дистрибутива fedora Linux и другими.

Управление версиями в вашем проекте

Ниже приводится простой пример использования системы управления версиями для создания репозитория проекта на платформе Django. В данном примере используется система Mercurial (команда hg).

Начнем с создания заготовки проекта на платформе Django.

```
$ django-admin.py startproject stuff_dev_site
$ cd stuff_dev_site
$ ./manage.py startapp stuff_app
$ ls stuff_app
__init__.py   manage.py    settings.pyc urls.py
__init__.pyc  settings.py  stuff_app
```

Теперь превратим рабочий каталог в репозиторий.

```
$ hg init
```

И создадим простой файл .hgignore (его можно создать либо с помощью команды echo, как показано ниже, либо с помощью текстового редактора), который сообщит системе Mercurial о том, что необходимо игнорировать все файлы .pyc с байт-кодом – они автоматически создаются компилятором Python и потому не нуждаются в управлении.

```
$ echo "\.pyc$" > .hgignore
```

По умолчанию файл .hgignore содержит шаблоны, являющиеся регулярными выражениями. Теперь можно пойти дальше и добавить файлы:

```
$ hg add
adding .hgignore
adding __init__.py
adding manage.py
adding settings.py
adding stuff_app/__init__.py
adding stuff_app/models.py
adding stuff_app/views.py
adding urls.py
```

Затем мы отправляем изменения с помощью команды `hg commit` и проверяем, были ли они записаны, с помощью команды `hg log`.

```
$ hg commit -m "Initial version of my project"
No username found, using 'pbx@example.org' instead
$ hg log
changeset:   0:e991df3d3205
tag:         tip
user:        pbx@example.org
date:        Sun Oct 07 13:49:14 2008 -0400
summary:     Initial version of my project
```

Каждый раз, когда изменения передаются в репозиторий Mercurial, система записывает набор изменений, присваивая ему идентификатор, состоящий из двух чисел: автоматически увеличивающегося целого числа, действительного только для данного репозитория, и шестнадцатеричного хеша, уникально идентифицирующего данный набор изменений независимо от того, помещался ли он в репозиторий или извлекался из него.

Давайте внесем простое изменение, проверим его и отправим в репозиторий.

```
$ echo "I_LIKE_CANDY = True" >> settings.py
$ hg status
M settings.py
$ hg commit -m "Apparently someone likes candy."
No username found, using 'pbx@example.org' instead
$ hg log
changeset:   1:65e7cda9f64b
tag:         tip
user:        pbx@example.org
date:        Sun Oct 07 13:57:53 2008 -0400
summary:     Apparently someone likes candy.

changeset:   0:e991df3d3205
user:        pbx@example.org
date:        Sun Oct 07 13:49:14 2008 -0400
summary:     Initial version of my project
```

Как только изменения будут записаны, они становятся доступны в виде «моментального снимка» проекта. Если вы вдруг решите, что все, что было сделано после набора изменений с номером 0, является ужасной ошибкой, то с помощью команды `hg revert --rev 0 --all` вы сможете восстановить состояние рабочего каталога, существовавшее на этот момент.

С этого момента вы можете отключиться и начать работу над своим проектом. Вы пишете программный код, проверяете его работу с сервером разработки и записываете изменения с краткими и осмысленными описаниями. Наконец пришло время развертывать проект.

Вы могли бы просто упаковать все файлы в архив, скопировать архив на сервер, в каталог, где должно находиться приложение, и извлечь файлы из архива. Все бы ничего, если бы вам не приходилось больше вносить изменения в свое приложение. Но, в предположении, что вы разработчик, а не увенчанный венком лауреат, вы наверняка захотите что-то исправить и улучшить, и постоянная ходьба по кругу «заархивировать-скопировать-разархивировать» – окажется слишком утомительной.

Было бы гораздо удобнее, если было бы возможно быстро получить копию репозитория; тогда с учетом того, что система Mercurial запоминает, откуда была получена копия, можно было бы последовательно извлечь любые изменения, которые применялись к оригиналу. Именно это делает команда `hg clone`.

Для простоты примера мы предполагаем, что развернутая копия веб-сайта находится на том же сервере, где хранится копия, находящаяся в разработке. Это означает, что для создания копии достаточно просто перейти в другой рабочий каталог и ввести команду `hg clone`, указав ей путь к оригинальному каталогу (где ведется разработка). Например, если мы находимся внутри оригинального каталога и нам требуется скопировать версию, находящуюся в разработке, в вышележащий каталог, мы могли бы выполнить следующие команды:

```
cd ..
$ hg clone stuff_dev_site stuff_live_site
8 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

В результате мы получим полную копию оригинального репозитория и рабочего каталога.

```
$ ls stuff_live_site/
__init__.py manage.py settings.py stuff_app urls.py
```

Минутку, а где же наши файлы `.pyc`? Мы сообщили системе Mercurial, что она должна игнорировать их, поэтому они не попали в репозиторий и отсутствуют в списке.

Теперь попробуем внести некоторые изменения в оригинальном каталоге (где ведется разработка).

```
$ cd stuff_dev_site
$ echo "I_LIKE_DOGS = True" >> settings.py
$ hg commit -m "Also, dogs are liked."
```

После того как изменения будут протестированы, мы можем добавить их в работающее приложение, для чего перейдем в каталог с работающим приложением и извлечем изменения из репозитория:

```
$ cd ../stuff_live_site
$ hg pull -u
pulling from /stuff_dev_site
searching for changes
```

```
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Мы можем использовать ту же самую команду `hg revert`, чтобы отменить внесение изменений на рабочем сайте. Например, это может потребоваться, если изменение, которое мы извлекли, неожиданно вызвало какие-то проблемы на работающем сайте, – несмотря на то, что мы протестировали их на сайте разработки. Каждая копия проекта имеет то же содержимое и ту же хронологию изменений.

Это был лишь поверхностный обзор, но хотелось бы надеяться, что он пробудил в вас интерес. Системы управления версиями вообще, и Mercurial в частности, обладают намного более широкими возможностями, чем мы продемонстрировали здесь. Дополнительные сведения, включая ссылки на превосходные бесплатные руководства по системам Subversion, Mercurial и другим, вы найдете на сайте *withdjango.com*.

Программное обеспечение управления проектами

Системы, управляющие версиями ваших исходных текстов, будут вам полезны, и некоторые разработчики умеют обходиться одной только системой управления версиями, и ничем больше. Однако масса других разработчиков используют семейство веб-приложений, которые играют вспомогательную роль при работе с системой управления версиями, обеспечивая не только веб-интерфейс к репозиторию с исходными текстами, но и давая возможность ведения списков заданий на будущее, создания документации и т. д.

Существует множество таких пакетов. Многие из них предоставляются в виде служб, таких как Google Code, SourceForge, Launchpad и Basecamp. Другие являются самостоятельными приложениями, которые вы можете использовать индивидуально. Одно из таких приложений, пользующихся популярностью у разработчиков открытого программного обеспечения, называется Trac; оно, что совершенно неудивительно, написано на языке Python.

Trac

Приложение Trac – это открытая система управления страницами wiki, слежением за ошибками, исходными текстами и проектами,, которая разрабатывается и поддерживается компанией Edgewall Software. Мы готовы признаться в своем расположении к ней: система Trac является для нас предпочтительным средством управления про-

ектами. Она отлично работает без дополнительных настроек и превосходно интегрируется с Subversion и другими системами управления версиями. Она использует широко распространенную разметку wiki для создания ссылок на версии программного кода, отчеты об ошибках и страницы с документацией и примечаниями.

Если вас это не убеждает, знайте, что даже репозиторий программного кода Django *http://code.djangoproject.com/* работает под управлением системы Trac. (Возможно, вы просто не распознали ее из-за отличных настроек стилей.)

Примечание

Система Trac была основным инструментом, применявшимся при работе над этой книгой, о чем дополнительно говорится в разделе «Выходные данные».

Следует также упомянуть, что хотя по умолчанию система Trac распространяется с поддержкой Subversion, тем не менее, с помощью расширений Trac возможно использовать другие системы (как те, что были описаны выше). Загрузить систему Trac и ознакомиться с документацией можно на сайте *http://trac.edgewall.org/*. Если у вас появится потребность в расширении возможностей этой системы, посетите сайт *http://trac-hacks.org/*, где вы найдете дополнительные расширения и другие наработки.

Текстовые редакторы

Для разработки проектов на платформе Django не требуется приобретать какое-либо специальное программное обеспечение. Подойдет любой текстовый редактор, оснащенный дополнительными функциями, обслуживающими создание программного кода. Ниже приводятся несколько рекомендаций для некоторых наиболее популярных редакторов.

Emacs

Наиболее важным элементом, необходимым, чтобы чувствовать себя довольным и продуктивным программистом Django, использующим Emacs, является расширение **python-mode**, которое обеспечивает подсветку синтаксиса, оформление отступов и некоторые другие особенности, упрощающие редактирование программного кода на языке Python. Emacs версии 22 и выше уже содержит встроенную версию расширения python-mode. Если вы используете более старую версию или разновидность Emacs, такую как XEmacs, обращайтесь на официальный веб-сайт Python (*http://www.python.org/emacs/*).

Существует также расширение, созданное пользователями, для редактирования шаблонов Django, которое можно найти в wiki Django (*http://code.djangoproject.com/wiki/Emacs*).

Vim

Vim – это текстовый редактор, основанный на старейшем текстовом редакторе vi в системе UNIX и обладающий невероятным числом расширений и улучшений по сравнению с оригинальным редактором. Загрузить этот редактор и поближе познакомиться с ним можно на веб-сайте *http://www.vim.org/*. Как и в случае с Emacs, существует расширение для редактора vim, обеспечивающее режим редактирования программного кода на языке Python. Исчерпывающий список советов по использованию редактора vim (и его разновидностей) для редактирования программного кода Python можно найти в wiki Django (*http://code.djangoproject.com/wiki/UsingVimWithDjango*).

TextMate

TextMate – это популярный коммерческий редактор для OS X, обладающий превосходной поддержкой Django. Поддержка синтаксиса различных языков программирования в редакторе TextMate организована в виде «пакетов», и в состав редактора уже входит пакет для языка Python, который ускоряет работу с программным кодом на языке Python и обеспечивает повышенную его удобочитаемость благодаря подсветке синтаксиса. Кроме того, в общедоступном репозитории пакетов редактора TextMate имеются два специализированных пакета для Django: один предназначен для работы с программным кодом Python, а второй – для работы с шаблонами Django. Дополнительную информацию можно найти на сайте wiki Django (*http://code.djangoproject.com/wiki/TextMate*).

Eclipse

Интегрированная среда разработки Eclipse предлагает мощный модуль разработки на языке Python, который называется PyDev. Получить и поближе познакомиться с модулем PyDev можно на странице проекта, на сайте SourceForge (*http://pydev.sourceforge.net/*).

D

Поиск, оценка и использование приложений на платформе Django

Вне всяких сомнений, создание приложений на платформе Django доставляет удовольствие, но вам едва ли захочется создавать все подряд с нуля. По мере роста популярности платформы вокруг нее сформировалась своя экосистема открытых приложений на платформе Django. Этому также способствует тот факт, что сама платформа Django является открытым проектом; это привело, например, к появлению этического правила, которое гласит, что, когда возникают сомнения, приложение следует выпускать как открытое.

Более того, многие из этих приложений, по примеру Django, выпускаются под неограничивающими лицензиями типа BSD/MIT. Это может упростить продажу таких приложений таким организациям, которые настороженно относятся к наиболее типичной открытой лицензии GNU Public License, или GPL. Одно из требований лицензии GPL заключается в том, что, если в приложении используется программный код, выпущенный под лицензией GPL, то в случае распространения приложения (в противоположность случаю, когда функциональность приложения предлагается как служба сети) его программный код должен быть открыт. Конечно, лицензии типа BSD/MIT тоже допускают открытие программного кода, но не требуют этого. Мы никак не оцениваем относительных достоинств этих двух подходов, но широко известно, что лицензии типа BSD/MIT являются более «дружественными для бизнеса», поэтому проблемы лицензирования могут повлиять на ваш выбор.

Что бы вам не потребовалось, – простое средство регистрации пользователей, что-то более сложное, такое как полнофункциональный механизм блога, или полноценное решение электронной коммерции для вашего интернет-магазина, – есть вероятность, что уже существует от-

крытое приложение на платформе Django, реализующее то, что необходимо. Но как найти такое приложение? Как оценить его? И как максимально упростить его использование (и обновление)?

Где искать приложения

Некоторые разработчики поддерживают свои собственные веб-сайты и/или собственные системы управления проектами, поэтому часто приложения на платформе Django можно найти в блогах их создателей или в личных экземплярах приложения Trac. Однако многие, если не большинство, разработчиков группируются на централизованных сайтах каталогов, где высока вероятность, что они будут замечены; и ниже мы перечислим некоторые из таких сайтов.

- Страница DjangoResources в wiki проекта Django: здесь многие авторы сообщают всему миру о своих приложениях. Прямая ссылка на страницу: *http://code.djangoproject.com/wiki/DjangoResources*.
- Google Code: бесплатное размещение и понятный интерфейс обеспечили сайту Google Code высокую популярность у разработчиков приложений на платформе Django. Если выполнить поиск по ключевому слову «django», то вы получите весьма длинный список.
- Djangopluggables.com: относительно новый (на момент написания этих строк) ресурс *djangopluggables.com* представляет собой сайт с удобным дизайном, на котором собрана информация о самых разнообразных приложениях на платформе Django.
- GitHub.com: популярный сайт, представляющий собой репозиторий исходных текстов и место общения, использующий систему управления версиями Git. Здесь можно найти несколько приложений Django, набирающих популярность вместе с системой Git.

По мере развития экосистемы приложений на платформе Django постоянно увеличиваются и возможности их поиска. Посетите сайт *withdjango.com*, где вы найдете постоянно обновляемый список ссылок и ряд рекомендаций.

Как оценивать приложения

Ниже приводятся несколько вопросов, ответы на которые помогут оценить найденное приложение или проект.

- Продолжается ли развитие? Когда были внесены последние изменения или исправления? Не каждый проект требует еженедельного обновления, но если последняя активность в работе над проектом наблюдалась год тому назад, то, скорее всего, он заброшен. Любое приложение, которое разрабатывалось для платформы Django с версией ниже 1.0, требует обновления, иначе оно быстро станет бесполезным.

- Имеется ли документация? *Действительно ли* это документация? Хорошо ли она организована и легко ли ее читать? Как много в нем такого рода предупреждений: «ПРИМЕЧАНИЕ: Этот раздел считается устаревшим!»? Похоже ли, что она активно поддерживается? (Например, обеспечивается ли доступ к документации посредством системы управления версиями и доступна ли она для загрузки?)
- Кто автор? Поищите в Google имя (имена) автора, чтобы получить представление о том, насколько автор опытен и как о нем отзываются в сообществе. Если, например, вы увидите множество одобрительных отзывов в почтовой рассылке проекта, это очень хорошо.
- Как написан программный код? Если вы обладаете некоторым опытом программирования на языке Python, самый лучший способ получить представление о проекте – это загрузить исходные тексты и просто заглянуть в них. Как организованы файлы с исходными текстами? Присутствуют ли в них строки документирования, поясняющие назначение функций и методов? Имеются ли наборы тестов и выполняются ли они?
- Имеется ли сообщество? Многие проекты на платформе Django начинались как приложения для удовлетворения потребностей некоторого человека или организации и со временем обретали своих пользователей и сообщества разработчиков. Чем сложнее приложение, тем более вероятно, что вокруг него существует сообщество опытных специалистов, которые могут оказать помощь в разрешении проблем. Чтобы считаться жизнеспособным, не каждое приложение должно иметь вокруг себя сплоченное сообщество, но определенная социальная структура должна иметься – пропорционально сложности приложения.

Как пользоваться приложениями

Сторонние приложения на платформе Django, так же, как и ваши, – это всего лишь модули на языке Python. Чтобы задействовать их в проекте, достаточно просто добавить строку, содержащую путь к приложению (в точечной нотации, принятой в языке Python) в параметр `INSTALLED_APPS`, в файле `settings.py` проекта.

Вы можете поместить приложение в любое место по своему желанию, но по сути у вас имеется три варианта:

- Встраивание: если приложение требуется задействовать в единственном проекте, вы можете сохранить его в каталоге проекта, рядом с другими приложениями. В некоторых случаях это самый простой вариант. Недостаток такого подхода проявится, если вы вдруг решите использовать это приложение в других проектах на том же сервере.
- Создание «разделяемого приложения»: другой вариант состоит в том, чтобы создать каталог для разделяемых приложений (его

можно назвать shared_apps) и добавить этот каталог в путь поиска Python. Это позволит хранить все сторонние приложения в одном месте и вместе с тем легко импортировать их в проектах. Чтобы добавить приложение в некоторый проект, нужно просто добавить имя приложения (здесь не требуется использовать точечную нотацию) в параметр INSTALLED_APPS настройки проекта.

- Установка в каталог site-packages: вы можете добавить приложение на платформе Django в системный каталог с библиотекой Python, который обычно называется site-packages. (Введите в интерактивной оболочке интерпретатора Python команду import sys; sys.path, чтобы точно узнать, какие каталоги находятся в пути поиска.)

Если вы следите за развитием проекта, вы можете извлекать его из системы управления версиями проекта, вместо того чтобы просто загружать файлы архивов. В этом случае вы извлекаете последнюю версию в свой проект, в каталог с разделяемыми приложениями или в каталог site-packages – в зависимости от того, какой из предыдущих вариантов вы выбрали. Но имейте в виду, что вы должны проявлять особую осторожность при обновлении подобных сторонних приложений и обязательно проверять, не была ли нарушена работоспособность проектов, использующих их.

Передача собственных приложений

Хотелось бы надеяться, что по мере вашего развития как разработчика приложений на платформе Django вы обнаружите, что некоторые из ваших разработок могут оказаться полезными для других. Мы призываем вас рассмотреть возможность выпуска таких приложений на условиях свободной лицензии, чтобы позволить другим пользователям Django использовать и развивать их. Такие службы размещения программного кода, как Google Code, SourceForge, GitHub и другие, помогут вам разместить свой проект (не создавая отдельный веб-сайт, который требует администрирования). Не забудьте сообщить об этом и нам!

E

Django и Google App Engine

В этом приложении рассказывается о переносе приложений Django на механизм Google App Engine – масштабируемую платформу веб-приложений, которая частично основана на платформе Django и реализует многие особенности Django. Мы постараемся охватить основы, но мы не сможем осветить все аспекты использования этой технологии. Ссылки на дополнительную информацию по этой теме вы найдете в конце приложения.

Существует несколько способов разработки приложений на основе механизма App Engine с различной степенью опоры на платформу Django:

- Новое приложение, опирающееся исключительно на использование App Engine
- Существующее приложение Django, перенесенное на платформу App Engine
- Новое приложение Django, написанное специально для использования App Engine

В первом случае используется минимальный круг возможностей и компонентов Django, присутствующих во всех приложениях на платформе App Engine, – мы рассмотрим их в следующем разделе. В двух других случаях на арену выходит платформа Django, и эти варианты станут предметом обсуждения данного приложения. Повторимся еще раз, если у вас появится желание создать новое приложение, опирающееся исключительно на использование App Engine, обращайтесь по ссылкам, что приводятся в конце, потому что здесь мы не будем рассматривать этот вариант.

Наша цель – разработка приложений на платформе Django, а рассказ об использовании Google App Engine заслуживает отдельной книги!

Назначение платформы App Engine

Появление механизма Google App Engine представляет особый интерес для разработчиков, использующих или планирующих использовать платформу Django. Его появление обусловило широкую поддержку языка Python и платформы Django внутри огромного сообщества разработчиков программного обеспечения и вызвало значительный интерес у тех, кто ранее даже не рассматривал возможность использования этих технологий.

Самым значительным достижением механизма App Engine является возможность беспроблемного его развертывания и сопровождения. Нам приходится выполнять настройку веб-серверов – Apache или Ngnix, с использованием `mod_python` или FCGI, и т. д., потому что нам это *необходимо* для создания веб-сайтов с использованием языка Python.

Такие сложности почти отсутствуют при использовании языка PHP, который, впрочем, страдает от множества других проблем, среди которых не самое последнее место занимают проблемы самого языка. Платформа App Engine предоставляет веб-разработчикам, использующим язык Python, подобный подход, лишенный многих проблем. Вдобавок ко всему, при этом появляется возможность использовать существующую (и весьма массивную) инфраструктуру Google.

Приложения, опирающиеся исключительно на использование App Engine

Приложения, опирающиеся исключительно на использование App Engine, создаются в единственном каталоге, и при их разработке используется инструментальный набор, предоставляемый платформой App Engine: веб-сервер разработки (`dev_appserver.py`), входящий в состав SDK (системный инструментарий разработчика), и средство выгрузки приложения (`appcfg.py`), используемое для развертывания в «облако» App Engine, когда ваш программный код будет готов к выпуску.

Конфигурация приложения на платформе App Engine определяется в виде файла формата YAML (`app.yaml`), содержимое которого может выглядеть примерно так:

```
application: helloworld
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
    script: helloworld.py
```

Обратите внимание на раздел `handlers`, который по своему назначению напоминает файлы URLconf в Django, связывая строки URL с опреде-

ленными сценариями. Еще одной особенностью платформы App Engine является система шаблонов, которая фактически основана на системе шаблонов Django.

Ограничения платформы App Engine

С точки зрения разработчика приложений на платформе Django, самым существенным недостатком реализации Google является отсутствие в ней отсутствие механизма ORM. Вместо реляционной базы данных компания Google задействовала свою собственную систему хранения данных BigTable (за дополнительной информацией обращайтесь по адресам *http://labs.google.com/papers/bigtable.html* и *http://en.wikipedia.org/wiki/BigTable*). Эта система не поддерживает ни отношений между данными, ни язык SQL, ни возможность преобразования в формат JSON и т. д.

Несмотря на то, что вы можете писать совершенно новые приложения, использующие другие компоненты платформы Django, а вместо обычных моделей данных подставлять механизм ORM компании Google, опирающийся на систему BigTable, тем не менее, этот недостаток практически исключает возможность использования существующих приложений на платформе Django. Безусловно, вы можете переписать части своих приложений, чтобы обойти это ограничение, но вы наверняка не будете делать этого в отношении приложений и компонентов платформы Django, таких как приложение администрирования, система аутентификации, универсальные представления и т. д.

Мы рассмотрим переработку ваших собственных приложений для использования совместно с платформой App Engine в следующем разделе.

В результате этих ограничений нам остается возможность использовать базовые возможности: файлы URLconf, представления и шаблоны. Этих компонентов вполне достаточно для создания любых веб-сайтов, однако они не способны полностью удовлетворить ваши потребности, если вы решите использовать существующие приложения Django совместно с App Engine или создавать приложения, которые можно было бы развертывать и как обычные приложения, и как приложения App Engine.

Примечание

К моменту написания этих строк в состав App Engine входили компоненты платформы Django, которые являются частью пусть и устаревшего, но стабильного выпуска Django (0.96.1).

Проект Google App Engine Helper для Django

Ключом к успешному освоению платформы App Engine является использование специального каркаса Google App Engine Helper для

Django, который делает ее чуть больше похожей на «настоящую» платформу Django. Это открытый проект, финансируемый компанией Google (кстати, имя Гвидо ван Россума (Guido van Rossum), создателя языка Python, значится в списке разработчиков этого проекта), цель которого состоит в том, чтобы превратить App Engine в комфортную среду для тех, кто уже имеет опыт работы с платформой Django. Он даже позволяет использовать более современные версии Django, чем та, что предлагается механизмом App Engine.

Получение SDK и Helper

Прежде чем двинуться дальше, необходимо получить необходимое программное обеспечение. Загрузить инструментарий Google App Engine SDK можно с домашней страницы проекта по адресу *http://code.google.com/p/googleappengine/*. Там можно найти установочный файл `.msi` для платформы Windows и файл `.dmg` – для Mac OS X. Для других платформ нужно просто загрузить архив в формате ZIP с исходными текстами. Точно так же пакет вспомогательного каркаса Helper можно загрузить по адресу *http://code.google.com/p/google-app-engine-django/*.

Установите оба пакета, следуя инструкциям. После установки можно приступать к экспериментам с платформой App Engine. Например, попробуйте, следуя обучающему руководству, создать простое приложение, опирающееся исключительно на эту платформу. Компания Google предоставляет неплохую документацию, описывающую, как установить и настроить App Engine, поэтому мы не будем воспроизводить ее здесь. Обучающее руководство App Engine можно найти по адресу: *http://code.google.com/appengine/docs/gettingstarted/*.

Совсем нелишним будет знать, как создавать простые приложения на обеих платформах, чтобы упростить понимание используемых подходов, когда мы будем пытаться объединять Google App Engine с существующим приложением на платформе Django или создавать новое приложение App Engine с уклоном в использование платформы Django.

Подробнее о Helper

К моменту написания этих строк проект Helper находился еще в зачаточном состоянии. Его главная цель состояла в том, чтобы скруглить острые углы платформы App Engine, существующие с точки зрения разработчика приложений на платформе Django. Но у этого проекта впереди еще долгий путь. Если заглянуть в исходные тексты Helper, можно узнать немного больше о внутреннем устройстве обеих платформ, App Engine и Django.

Каркас Helper не превращает App Engine *в* Django, но он способен облегчить переход. В примере, следующем ниже, предполагается использование каркаса Helper, тем не менее, ничто не мешает использовать базовые функциональные возможности App Engine без него. Как

уже не раз упоминалось, «Django – это всего лишь программа на языке Python». Не воспринимайте острые углы App Engine как препятствие к исследованию возможностей этой платформы. По словам одного наблюдателя, платформа App Engine «усложняет создание немасштабируемых решений». Если вы готовы принять удобство в развертывании и обещания огромной масштабируемости в обмен на перечисленные выше ограничения, тогда продолжайте читать.

Каркас Helper представляет собой заготовку проекта на платформе Django, содержащую единственное приложение с именем `appengine_django`. Эту заготовку можно использовать как основу для нового проекта или скопировать папку с приложением `appengine_django` в существующий проект.

Платформа App Engine предоставляет механизм администрирования с минимальными возможностями, который называется Development Console. После установки каркаса Helper это приложение администрирования будет доступно по адресу URL *http://localhost:8000/_ah/admin* (при условии, что сервер App Engine использует порт 8000 для приема соединений). В отличие от обычного приложения администрирования этот инструмент способен работать только с моделями, для которых в хранилище данных уже имеется хотя бы одна запись. Если у вас имеется модкль, для которой еще не было создано ни одной записи, вы не сможете использовать Development Console для создания записей этого типа.

Интегрирование App Engine

В этом разделе мы возьмем за основу наш простой блог, разработанный в главе 2 «Django для нетерпеливых: создание блога», и превратим его в приложение на платформе App Engine. Мы пройдем все этапы, описанные в файле README проекта Google App Engine Helper для Django, который находится по адресу *http://code.google.com/p/google-appengine-django/source/browse/trunk/README*. В следующем примере предполагается, что проект называется `mysite`, а приложение – `blog`.

Копирование программного кода App Engine в проект

После загрузки и распаковывания архива с каркасом Helper у вас появится каталог `appengine_helper_for_django`. Скопируйте содержимое этого каталога в корневой каталог существующего проекта блога на платформе Django, чтобы файлы `app.yaml`, `main.py` и каталог `appengine_django` находились в одном каталоге с файлами `urls.py` и `settings.py`.

Теперь отредактируйте файл `app.yaml` и измените имя приложения на то, под которым оно будет зарегистрировано в приложении Admin Console платформы App Engine. Следующий этап состоит в том, чтобы обеспечить вашему проекту доступ к программному коду самой платформы App Engine.

Примечание

В действительности изменять имя приложения, под которым оно будет зарегистрировано в платформе App Engine, совсем необязательно. Это необходимо, только если вы решите выгрузить свое приложение в App Engine. Ничто не препятствует возможности запускать свой программный код на сервере разработки, входящем в комплект SDK, без ввода этой информации.

Если вы установили App Engine SDK не с помощью инсталлятора для Windows или Mac OS X, то необходимо вручную добавить ссылку на программный код App Engine. В POSIX-совместимых системах (Linux или Mac OS X) вызовите команду `ln`, которая должна выглядеть примерно так, как показано ниже:

```
$ ln -s THE_PATH_TO/google_appengine ./.google_appengine
```

Интегрирование App Engine Helper

Наш следующий шаг – интегрирование каркаса Helper в утилиту `manage.py`. Это позволит нам управлять нашим приложением практически так же, как обычным приложением на платформе Django. Для этого нужно добавить две новые строки в самое начало файла (после строки «she-bang») непосредственно перед инструкцией импортирования модуля `execute_manager`, то есть первые четыре строки в файле `manage.py` должны выглядеть так:

```
#!/usr/bin/env python

from appengine_django import InstallAppengineHelperForDjango
InstallAppengineHelperForDjango()

from django.core.management import execute_manager
```

Это даст механизму App Engine возможность взять частичный контроль над некоторыми командами управления, а также добавлять программный код App Engine и обеспечить доступность дополнений, входящих в состав каркаса Helper. Однако даже после модификации файла `manage.py` вы сохраняете доступ ко многим важнейшим командам.

Теперь, «обновив» утилиту `manage.py`, попробуем использовать ее для создания нового файла `settings.py`. (Мы рекомендуем сохранить резервную копию оригинала на случай, если вы пожелаете вернуть все назад!) Зачем это необходимо? К сожалению, мы вынуждены удалить все функциональные возможности Django, несовместимые с App Engine, – все, что было описано выше. Для этого необходимо воспользоваться командой `diffsettings`.

```
$ manage.py diffsettings
WARNING:root:appengine_django module is not listed as an application!
INFO:root:Added 'appengine_django' as an application
WARNING:root:DATABASE_ENGINE is not configured as 'appengine'. Value
overridden!
```

```
WARNING:root:DATABASE_%s should be blank. Value overridden!
WARNING:root:Middleware module 'django.middleware.doc.XViewMiddleware'
is not compatible. Removed!
WARNING:root:Application module 'django.contrib.contenttypes' is not
compatible. Removed!
WARNING:root:Application module 'django.contrib.sites' is not compatible.
Removed!
DATABASE_ENGINE = 'appengine'
DEBUG = True
INSTALLED_APPS = ('django.contrib.auth', 'django.contrib.admin',
'mysite.blog', 'appengine_django')
MIDDLEWARE_CLASSES = ('django.middleware.common.CommonMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware')
ROOT_URLCONF = 'mysite.urls'  ###
SECRET_KEY = 'w**sb^(p($wzxra*a9_@4_zOs(9i(9x3(w-aribbaaa4(r^wi'
SERIALIZATION_MODULES = {'xml': 'appengine_django.serializer.xml'}  ###
SETTINGS_MODULE = 'settings'  ###
SITE_ID = 1  ###
TEMPLATE_DEBUG = True
TIME_ZONE = 'America/Los_Angeles'
```

Все строки в выводе команды, начинающиеся со слова «WARNING», обычно указывают на то, что был удален некоторый несовместимый или некорректный параметр настройки. Все строки, следующие *после* строк со словами «WARNING» и «INFO», представляют собой содержимое нового файла settings.py.

Перенос приложения на платформу App Engine

Теперь нам необходимо изменить свое приложение, чтобы задействовать в нем объекты платформы App Engine, и в первую очередь – файл models.py. Вместо класса django.db.models.Model мы должны использовать класс appengine_django.models.BaseModel. Создайте резервную копию оригинального файла models.py и измените его, как показано ниже:

```
from appengine_django.models import BaseModel
from google.appengine.ext import db

class BlogPost(BaseModel):
    title = db.StringProperty()
    body = db.StringProperty()
    timestamp = db.DateTimeProperty()
```

Общие черты определенно сохранились – объекты этого класса эквивалентны тем, что имелись раньше. Одно заметное отличие состоит в том, что теперь отпала необходимость указывать параметр max_length для поля title. Поле типа StringProperty в платформе App Engine может хранить строки длиной до 500 байтов. Для простоты мы использовали тот же тип для свойства body.

Если есть вероятность, что объем одного сообщения в блоге может превысить этот размер, то можно использовать поле типа TextProperty. По-

ля этого типа могут хранить более 500 байтов, но они не индексируются и не могут использоваться для фильтрации или участвовать в сортировке. Конечно, вы скорее всего и не собирались фильтровать или сортировать содержимое блога по атрибуту body, поэтому в данном случае это не будет большой потерей.

Так как платформа App Engine не использует механизм ORM Django, вам придется изменить реализацию функции представления, чтобы использовать в ней запросы BigTable. При этом вы обнаружите, что прикладной интерфейс к базе данных, реализованный в App Engine, обеспечивает два различных способа запросить данные: с помощью стандартного объекта типа Query или с помощью объекта типа GqlQuery. В нашем случае мы предпочли использовать подход, который ближе к ORM, сохранив возможность доступа к данным на высоком уровне, поэтому мы просто используем объекты типа Query.

Опробование

Итак, все готово к опробованию нашего перестроенного приложения. Теперь при запуске сервера разработки командой manage.py runserver вы больше не увидите информацию о проверке моделей, имя используемого файла с настройками и строку с текстом начального сообщения. Вместо этого вы увидите вывод сервера App Engine.

```
$ manage.py runserver
INFO:root:Server: appengine.google.com
INFO:root:Running application mysite on port 8000: http://localhost:8000
```

Откройте в броузере страницу с адресом URL *http://localhost:8000/blog*. Вы должны увидеть уже знакомую начальную страницу блога, как показано на рис. Е.1. Конечно, в ней отсутствуют записи, потому

Рис. Е.1. В «новом» приложении блога пока нет записей

что сейчас – вместо механизма ORM платформы Django и базы данных – мы используем хранилище платформы App Engine, в котором еще ничего нет!

Добавление данных

Как уже упоминалось выше, приложение администрирования платформы Django недоступно при использовании платформы App Engine. Не имея приложения администрирования, нам придется добавить первую запись вручную (с помощью интерактивной оболочки интерпретатора Python), чтобы механизм ввода данных платформы App Engine смог распознать конкретную модель. На этой машине у нас установлена оболочка IPython, поэтому в примере можно наблюдать начальное сообщение и приглашения к вводу этой оболочки.

```
$ manage.py shell
Python 2.5.1 (r251:54863, Mar 7 2008, 04:10:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.2 - An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
```

Получив доступ к строке с приглашением к вводу, мы можем импортировать класс `BlogPost`, создать его экземпляр и заполнить его поля.

```
In [1]: from blog.models import BlogPost
In [2]: entry = BlogPost()
In [3]: entry.title = '1st blog entry'
In [4]: entry.body = 'this is my 1st blog post EVAR!!'
In [5]: from datetime import datetime
In [6]: entry.timestamp = datetime.today()
```

Поля класса `DateTimeProperty` в платформе App Engine принимают объекты `datetime.datetime`, поэтому мы импортировали и использовали соответствующую библиотеку языка Python. Затем с помощью функции `today` мы получили текущую дату и время и присвоили это значение полю `timestamp`. Теперь, чтобы сохранить объект, достаточно просто вызвать его метод `put`.

```
In [7]: entry.put()
Out[7]: datastore_types.Key.from_path('BlogPost', 1, _app=u'mysite')
In [8]: query = BlogPost.all()
In [9]: for post in query:
    ....: print post.title, ':', post.body, '(%s)' % post.timestamp
    ....:
1st blog entry : this is my 1st blog post EVAR!! (2008-07-13 12:28:52.140000)
```

Как отмечается в документации с описанием API базы данных: «Метод `all()` класса `Model`... возвращает объект `Query`, представляющий за-

прос всех записей соответствующего вида.». (Эта цитата взята из документации, описывающей приемы манипулирования данными, которую можно найти по адресу *http://code.google.com/appengine/docs/datastore/creatinggettinganddeletingdata.html*.) Как видите, мы сумели получить недавно добавленные данные внутри оболочки. Теперь нам осталось замкнуть круг и просмотреть добавленные данные в нашем приложении.

Убедитесь, что сервер разработки еще работает, и затем обновите в веб-броузере страницу с адресом URL *http://localhost:8000/blog*. Теперь в блоге должна появиться запись, как показано на рис. Е.2.

Хотелось бы надеяться, что этот короткий пример помог вам выше, чем прежде, оценить приложение администрирования, входящее в состав платформы Django. Кроме того, обратите также внимание, что интерактивная оболочка интерпретатора Python может оказаться ценным инструментом для работы с данными, который не зависит от состояния вашего веб-сайта.

Рис. Е.2. Теперь в нашем блоге появилась запись!

Создание нового приложения на платформе Django, использующего возможности App Engine

Чтобы перенести приложение с платформы Django на платформу App Engine, вам в действительности придется решить более сложную задачу. Создание нового приложения выглядит значительно проще, потому что при этом у вас нет «багажа» существующего приложения на платформе Django. Шаги по созданию нового приложения Django, ис-

пользующего возможности App Engine, приведенные ниже, будет выполнять проще, если учитывать сведения из предыдущего раздела.

1. Как обычно, создать с помощью сценария `django-admin.py` новый проект Django.
2. Скопировать программный код App Engine (`app.yaml`, `main.py`, `appengine_django`) в каталог проекта.
3. Отредактировать файл `app.yaml`, заменив имя приложения (которое будет зарегистрировано в приложении App Engine Admin Console).
4. Если необходимо, добавить ссылку на программный код App Engine.
5. Выполнить команду `manage.py startapp ИМЯ_НОВОГО_ПРИЛОЖЕНИЯ`.
6. Создать приложение!

При создании своего приложения необходимо помнить, что создается не «чистое приложение на платформе Django», – то есть не забывать использовать модели каркаса App Engine Helper вместо моделей Django. В качестве бонуса вы получаете доступ ко всей мощи прикладных интерфейсов App Engine:

- Среда выполнения для языка Python
- Интерфейс доступа к хранилищу данных
- Интерфейс для работы с изображениями
- Интерфейс для работы с электронной почтой
- Интерфейс для работы с механизмом кэширования Memcache
- Интерфейс для работы с адресами URL
- Интерфейс управления пользователями

Мы рекомендуем вам ознакомиться с документацией App Engine, чтобы понять, какие возможности предоставляют эти платформы.

В заключение

В этом приложении мы представили вам платформу Google App Engine, представляющую собой альтернативную платформу разработки приложений Django, и познакомили с некоторыми ее возможностями. Однако в обмен на новые функциональные возможности придется пожертвовать некоторыми особенностями платформы Django, что может вызывать определенные сложности при переходе на платформу App Engine для программистов, привыкших использовать Django. К счастью, вспомогательный каркас, разработанный для App Engine, способен немного облегчить такой переход.

Затем мы подробно рассмотрели порядок перевода существующего приложения на платформу App Engine. Создании нового приложения с самого начала требует выполнения сходных этапов, за исключением того, что в этом случае не нужно беспокоиться по поводу уже существовавшего программного кода. В качестве упражнения в овладении

App Engine мы предлагаем вам создать приложение блога с самого начала, реализовав его как 100-процентное приложение на платформе App Engine.

Главное для вас как для разработчика приложений на платформе Django заключается в том, что вы можете по-прежнему писать программы с использованием Django и при этом иметь в своем распоряжении преимущества, предлагаемые платформой App Engine: простое развертывание, свободную масштабируемость и возможность использовать существующую инфраструктуру Google.

Ресурсы в Интернете

Ниже приводится список важных ресурсов, которыми мы хотели бы поделиться с вами. Более исчерпывающий список вы найдете на веб-сайте книги *withdjango.com*.

- Google App Engine

  http://code.google.com/appengine/

- Проект App Engine SDK

  http://code.google.com/p/googleappengine/

- Обучающее руководство App Engine

  http://code.google.com/appengine/docs/gettingstarted/

- Проект Google App Engine Helper для Django

  http://code.google.com/p/google-app-engine-django/

- Использование каркаса Google App Engine Helper для Django (Мэтт Браун (Matt Brown), май 2008)

  http://code.google.com/appengine/articles/appengine_helper_for_django.html

Видеоматериалы

- Быстрая разработка с использованием Python, Django и Google App Engine (Гвидо ван Россум (Guido van Rossum), май 2008)

  http://sites.google.com/site/io/rapid-development-with-python-django-and-googleapp-engine

- Введение в Google App Engine на сайте Google Campfire (разные авторы, апрель 2008, 7 видеороликов)

  http://innovationstartups.wordpress.com/2008/04/10/google-app-engine-youtubes/

F

Участие в проекте Django

Django – это не просто платформа для разработки веб-приложений. Это не просто крупная архитектура и 50 000 строк кода. Это еще и сообщество программистов, тестеров, переводчиков, тех, кто отвечает на вопросы, и обширный коллектив добровольцев. В файле AUTHORS платформы Django содержится более 200 имен разработчиков, но кроме этого существует великое множество других, кто внес свой посильный вклад в развитие проекта.

Django – это яркий образец проекта с открытыми исходными текстами, как сказал автор языка Python, Гвидо ван Россум (Guido van Rossum); а это, кроме всего прочего, означает, что проект предлагает множество способов участия в нем. Вполне вероятно, что вы, как и многие другие, обнаружите, что платформа Django сделала вашу жизнь как веб-разработчика легче и разнообразнее; и найдете свободное время и желание принять участие в проекте. Ниже приводятся несколько примеров вашего участия.

В самых простых случаях от вас даже не потребуется писать программный код:

- Вы можете присылать исправления опечаток, обнаруженных в документации
- Вы можете помогать отвечать на вопросы вновь прибывших на каналах IRC или в списке рассылки django-users
- Вы можете помогать в сортировке сообщений об ошибках на сайте code.djangoproject.com, закрывая неверные сообщения и выясняя (или проверяя) подробности по верным сообщениям

Если у вас появится желание приложить руки, имеется несколько вариантов участия, связанных с программированием, которые не потребуют от вас титанических усилий:

- Присылать исправления известных ошибок
- Тестировать Django на наименее популярных системах или в необычных (но поддерживаемых) конфигурациях, чтобы помочь в выявлении потенциальных проблем
- Присоединиться к работе в «ветви» Django, где разрабатываются новые особенности

Наконец, в разряде титанических усилий, – решение следующих задач оказало бы существенную помощь сообществу Django:

- Выполнить локализацию для языка, перевод на который еще отсутствует в проекте (если вам удастся отыскать такой!)
- Найти особенность, которая была бы желательна для многих, но пока не реализована, и реализовать ее
- Создать высококачественное приложение на платформе Django и выпустить его с открытыми исходными текстами

Если вы не пока не выбрали, каким из предложенных советов воспользоваться, присоединяйтесь к спискам рассылки **django-users** и **django-developers** или проведите некоторое время на канале IRC #django, на сайте *irc.freenode.net*. В самое короткое время вы выясните, чем могли бы помочь. А если вы встретите там кого-нибудь из нас, скажите: «Привет!».

Примечание

Документация Django (на официальном сайте *withdjango.com*) достаточно подробно охватывает вопросы участия в проекте, начиная от стиля оформления программного кода и заканчивая описанием протокола списка рассылки.

Кроме присоединения к спискам рассылки и/или подключения к каналам IRC существует еще один способ присутствия в экосистеме Django – зарегистрироваться на сайте *http://djangopeople.net*, всемирном каталоге разработчиков Django (который также может использоваться для поиска других разработчиков, если вы будете заинтересованы в формировании группы специалистов для разработки проекта). Существует еще один сайт, стремящийся стать связующим звеном между разработчиками и теми, кто нуждается в их услугах, – *http://djangogigs.com*, где также имеется свой список разработчиков (*http://djangogigs.com/developers/*).

Наконец, вам, скорее всего, стоит подписаться на получение новостей в официальном блоге Django, что может стать отличным способом следить за событиями, происходящими в мире Django. Агрегатор новостей, а также массу ссылок на другие инструменты, используемые сообществом, можно найти по адресу: *http://djangoproject.com/community/*.

Алфавитный указатель

Специальные символы

(символ решетки), комментарии, 37
$ (знак доллара)
  в файлах URLconf, 172
  переменная (jQuery), 283
% (знак процента), оператор форматирования строк, 53
& (амперсанд) оператор (Django), 161
* (звездочка)
  в функциях (Python), 75
  дублирование, 45
** (две звездочки) в функциях (Python), 75
+ (знак плюс), конкатенация, 45
, (запятая) в кортежах, 55
: (двоеточие) извлечение срезов последовательностей, 44
<, оператор перенаправления, 375
^ (крышка) в файлах URLconf, 172
| (вертикальная черта)
  оператор (Django), 374
| (вертикальная черта) оператор (Django), 161
~Q, в комбинации с именованными аргументами, 161
>, оператор перенаправления, 375
>>, оператор добавления в конец файла, 375

A

ab (инструмент измерения производительности), 344
Ajax (Asynchronous JavaScript And XML – асинхронный JavaScript и XML), 271
  XML или JSON, 280
  библиотеки (пример приложения liveblog)
    выбор, 274, 281
    настройка и тестирование, 282
  описание, 272
  преимущества, 272
  требования к использованию, 279
all, функция (Python), 56
any, функция (Python), 56, 57
Apache, веб-сервер
  обслуживание статических файлов, 393
  подключение Django, 391
  установка, 390
append, метод (Python), 47
AssertionError, исключение (Python), 65
AttributeError, исключение (Python), 65
AutoField, класс (Django), 140

B

base_site.html, шаблон (приложение администрирования), расширение, 311
BigTable, система хранения данных, 418
block, тег шаблона (Django), 116
blog, пример проекта
  запуск сервера разработки, 98
  модели
    создание, 101
  настройка базы данных, 101
  приложение администрирования
    настройка, 105
    регистрация, 108
  создание приложения, 100
  создание проекта, 96
  шаблоны URL
    создание, 112, 114
BooleanField, класс (Django), 139

C

cache, тег шаблонов (Django), 351
CACHE_BACKEND, параметр настройки, 353
cache_control, декоратор (Django), 349
cache_page, декоратор (Django), 348
calendar, модуль (Python), 331
capitalize, метод (Python), 51
CharField, класс (Django), 138
CherryPy, веб-сервер, 394
class, ключевое слово (Python), 80
clear, метод (Python), 60
complex, тип данных (Python), 41
cookies, 178
COOKIES, структура данных (Django), 178
count, метод (Python), 51
create_update.create_object, универсальное представление (Django), 186
create_update.update_object, универсальное представление (Django), 186
CSS (Cascading Style Sheets – каскадные таблицы стилей), 122
CVS (Concurrent Versions System – система управления параллельными версиями), 403
Cygwin, 368

D

DATABASE_ENGINE, параметр, 102
DATABASE_HOST, параметр, 102
DATABASE_NAME, параметр, 102
DATABASE_PASSWORD, параметр, 102
DATABASE_PORT, параметр, 102
DATABASE_USER, параметр, 102
date_based.*, универсальные представления (Django), 186
db, механизм кэширования, 353
Decimal, тип данных (Python), 41
del, команда (Python), 59
delete, метод (Django), 229
  порядок выполнения операций, 229
Development Console, 420
distinct, метод (Django), 159
Django
  block, тег шаблона, 116
  for, тег шаблона, 113
  Syndication, приложение, 314
    настройка лент новостей, 314
    определение адреса URL ленты, 316
  адреса URL, 170
    include, функция, 174
    url, функция, 172
    URLconf, файл, 171
    вызываемые представления, 175
    несколько объектов patterns, 173
  библиотеки тегов, 328
  блог агрегатор новостей, 429
  версия в разработке, 388
  вспомогательные сценарии, 339
    задания cron, 340
    импорт/экспорт данных, 341
  встроенный сервер разработки, 390
  декораторы
    cache_control, 349
    cache_page, 348
    never_cache, 349
    stringvalue, 335
  документация, 429
  загружаемые файлы, генерирование, 317
    vCard, 319
    вывод диаграмм и графиков, 321
    значения, разделенные запятыми (CSV), 320
    конфигурационные файлы Nagios, 318
  загрузка
    версии в разработке, 388
    официального выпуска, 388
  конфигурационные файлы (Nagios), генерирование, 318
  кэширование, 296, 343
    кэширование всего сайта, 347
    объектов, 350
    определение производительности сервера, 344
    стратегии, 347
      cache, тег шаблонов, 351
      кэширование отдельных представлений, 348
      управление заголовками, имеющими отношение к кэшированию, 348

Django
кэширование
тестирование, 346
типы механизмов кэширования, 352
установка типа кэширования, 345
менеджеры, 325
модели
Meta, класс, 149
ORM (Object-Relational Mapper – объектно-реляционное отображение), 135
абстрактные базовые классы, 146
безопасность, 137
внешние ключи, 140
выразительность, 137
и классы (Python), 83
инкапсуляция методов, 136
использование возможностей SQL, не предоставляемых платформой Django, 164
многотабличное наследование, 148
наследование, 145
ограничение отношений, 145
определение, 135
отношения между моделями, 140
параметры модели в приложение администрирования, 151
первичные ключи, 139
переносимость, 136
регистрация, 106
система управления содержимым, пример приложения, 248
тестирование, 359
типы полей, 137
уникальные значения, 139
фотогалерея, пример приложения, 218
модель запрос-ответ HTTP, 176
объекты запросов, 177
объекты ответов, 180
промежуточная обработка, 181
обработка исключений, 66
общий обзор архитектуры, 126, 128
оснастки, 105, 166
основные принципы, 128
DRY (Don't Repeat Yourself – не повторяйся), 129
быстрая разработка, 130

Django
основные принципы
слабая зависимость и гибкость, 129
официальные выпуски, 388
подключение к веб-серверу Apache, 391
представления, 183
как функции на языке Python, 183
собственные представления, 188
универсальные представления, 184, 290
приложение администрирования, 307
использование декораторов аутентификации, 312
настройка, 105, 307
настройка, с помощью параметра fieldsets, 308
расширение базовых шаблонов, 310
регистрация, 108
создание представлений, 312
приложения
liveblog, пример приложения
XML или JSON, 280
выбор библиотеки Ajax, 274
использование функции представления в JavaScript, 286
настройка и тестирование библиотеки Ajax, 282
создание функции представления, 284
структура каталогов, 275
шаблоны, 276
установка библиотеки Ajax, 281
Pastebin, пример приложения, 290
models.py, файл, 291
ограничение числа отображаемых записей, 300
определение адресов URL, 294
очистка с помощью задания cron, 302
подсветка синтаксиса, 301
создание шаблонов, 293
тестирование, 296
инструменты тестирования, 361

Django
приложения
интегрирование App Engine, 420
интегрирование App Engine Helper, 421
оценка, 413
передача, 415
перенос на платформу App Engine, 422
поиск, 413
проектирование, 273
тестирование, 356
doctest, модуль (Python), 357
unittest, модуль (Python), 358
установка, 414
фотогалерея, пример приложения, 217
models.py, файл, 218
программный код
настройка, 343
тестирование, 362
промежуточная обработка запросов, 182
промежуточная обработка ответов, 182
словари, 59, 295
создание проекта, 96
создание учетной записи суперпользователя, 104
специализированные менеджеры, 323
методы, именование, 325
методы, создание, 325
специализированные фильтры, 333
тег блока, 113
установка, 388
тестирование, 389
участие в проекте, 428
фильтры, 300
формы, 199
виджеты, 211
виджеты по умолчанию, 212
заполнение, 205
несвязанные, 205
нормализация данных, 209
определение, 199
основанные на моделях, 200
отображение, 209
проверка, 207
связанные, 205
создание подклассов форм, 204
Django
шаблоны, 112, 191
liveblog, пример приложения, 276
базовый шаблон, 115
блочные теги, 196
включение, 198
внедрение операций JavaScript, 283
для приложения Flatpages, 245
контекст, 192
наследование, 198
пример приложения системы управления содержимым, 261
расширение, 326
сиснтаксис, 193
создание, 112
Pastebin, пример приложения, 293
специализированные теги, 326
специализированные фильтры, 333
теги, 191, 195
теги включения, 330
фильтры, 118, 191, 194
фотогалерея, пример приложения, 235
шаблоны URL, 114
создание, 114
django-admin.py, утилита (Django), 96
djangogigs.com, 429
djangopeople.net, 429
Djangopluggables.com, 413
DjangoResources, страница wiki проекта Django, 413
django.shortcuts, модуль (Python), 188
__doc__, атрибут (Python), 93
doctest, модуль (Python), 357
Dojo, библиотека JavaScript, 274
dpaste.com, сайт, 291
DRY (Don't Repeat Yourself – не повторяйся), принцип, 129
DRY (Don't Repeat Yourself – не повторяйтесь), принцип, 101
dummy, механизм кэширования, 353
dumpdata, команда (Django), 154

E

Easy Install (дополнение для Python), 386
Eclipse, среда разработки, 411

elif, инструкция (Python), 61
else, инструкция (Python), 61
Emacs, текстовый редактор, 410
EmailField, класс (Django), 138
encoding, переменная (объекты запросов), 180
endswith, метод (Python), 51
enumerate, функция (Python), 56, 62
в моделях платформы Django, 62
eval, функция (Python), 56
extend, метод (Python), 88
extra, метод (Django), 162

F

False, значение (Python), 39
FastCGI, 395
feeds.py, файл, настройка, 314
Field, класс (Django), 138
fielfsets, параметр (приложение администрирования), 308
file, механизм кэширования, 353
FileField, класс (Django), 139
FilePathField, класс (Django), 139
FILES, структура данных (Django), 180
find, метод (Python), 51
Flatpages, приложение, 243
включение, 244
обработка ошибок, 245
создание шаблонов, 245
тестирование, 246
float, тип данных (Python), 40, 41
flup, модуль, 395
for, тег шаблона (Django), 113
for, цикл (Python), 61
ForeignKey, класс (Django), 140
Form, класс (Django), 199
fromkeys, метод (Python), 60

G

get, метод (Python), 58, 60
GET, структура данных (Django), 177
getJSON, метод (jQuery), 287
get_list_or_404, функция (Django), 188
get_object_or_404, функция (Django), 188
Git, система управления версиями
краткий обзор, 406
GitHub.com, 413
Google App Engine, 416
обучающее руководство, 419
ограничения, 418
перенос приложений Django на платформу App Engine, 420
добавление данных, 424
доступ к объектам App Engine, 423
копирование программного кода App Engine, 420
опробование, 423
преимущества использования, 417
приложения
создание приложений на платформе App Engine, 425
Google App Engine Helper for Django, проект, 418
загрузка, 419
Google App Engine SDK, загрузка, 419
Google Code, 413
group, функция (Python), 84
groups, функция (Python), 84

H

help, функция (Python), 93
HTML (HyperText Markup Language – язык разметки гипертекста), 122
HTTP (HyperText Transfer Protocol – протокол передачи гипертекста), 121
HTTP, модель запрос-ответ (Django), 176
объекты запросов, 177
объекты ответов, 180
промежуточная обработка, 181
Http404, исключение, возбуждение, 66
Http404, класс (Django), 188
HttpRequest, класс (Django), 177
HttpResponse, класс (Django), 180

I

IBM DB2, база данных, 400
if, инструкция (Python), 61
ImageField, класс (Django), 227
проверка, 222
ImageFieldFile, класс (Django), 227
ImportError, исключение (Python), 65
in, оператор (Python), 45
include, тег (Django), 199
include, функция (Django), 174

IndentationError, исключение (Python), 65
index, метод (Python), 51
IndexError, исключение (Python), 65
__init__, метод инициализации (Python), 81, 90
__init__.py, файл, 97
insert, метод (Python), 47
INSTALLED_APPS, кортеж, правак, 100
int, тип данных (Python), 40, 41
IOError, исключение (Python), 65
IPAddressField, класс (Django), 138
IPython (дополнение для Python), 386
islower, метод (Python), 51
isupper, метод (Python), 51
is_valid, метод (Django), 208
items, метод (Python), 60
iteritems, метод (Python), 60
iterkeys, метод (Python), 60
itervalues, метод (Python), 60

J

JavaScript, 122
join, метод (Python), 51
jQuery, библиотека JavaScript, 274
  настройка и тестирование, 282
  установка, 281
JSON (JavaScript Object Notation – формат записи объектов JavaScript), 122, 280
  функциональная совместимость с Python, 281

K

KeyboardInterrupt, исключение (Python), 65
KeyError, исключение (Python), 65
keys, метод (Python), 60

L

lambda, ключевое слово (Python), 73
  в декораторах аутентификации, 313
len, функция (Python), 56, 57
lighttpd, веб-сервер, 394
limit_choices_to, аргумент (Django), 145
Linux
  установка Python, 381
list, функция (Python), 56
list_detail.object_detail, универсальное представление (Django), 185
list_detail.object_list, универсальное представление (Django), 185
liveblog, пример приложения
  XML или JSON, 280
  выбор библиотеки Ajax, 274
  использование функции представления в JavaScript, 286
  настройка и тестирование библиотеки Ajax, 282
  проектирование, 273
  создание функции представления, 284
  структура каталогов, 275
  установка библиотеки Ajax, 281
  шаблоны, 276
liveblog, проект примера, 271
load, тег (шаблоны Django), 329
loaddata, команда (Django), 154
locmem, механизм кэширования, 353
long, тип данных (Python), 40
lower, метод (Python), 51
lstrip, метод (Python), 51

M

Mac OS X
  использование командной строки, 368
  установка Python, 381
Mako, механизм шаблонов, 337
manage.py shell, команда (Django), 36
manage.py, утилита, 97
  запуск сервера разработки, 98
  создание и изменение баз данных, 152
  создание приложения, 100
Manager, класс (Django), 154
ManyToManyField, класс (Django), 142
Markdown (легковесный язык разметки) в примере приложения системы упрвления содержимым, 252
match, функция (Python), 85
max, функция (Python), 56, 57
memcached, механизм кэширования, 353
Mercurial, система управления версиями
  краткий обзор, 405
Meta, класс (Django), 149

META, структура данных (Django), 180
method, переменная (объекты запросов), 179
Microsoft IIS, веб-сервер, 394
Microsoft SQL Server, база данных, 400
min, функция (Python), 56
MochiKit, библиотека JavaScript, 274
ModelChoiceField, класс (Django), 204
ModelForm, класс (Django), 200
models.py, файл, 101
  Pastebin, пример приложения, 291
  создание модели, 101
  фотогалерея, пример приложения, 218
mod_python, модуль, 390
  подключение Django, 391
mod_wsgi, модуль, 394
MooTools, библиотека JavaScript, 274
MVC (Model-View-Controller – модель-представление-контроллер), 123
  в платформе Django, 124
  модели, 125
  представления, 125
  шаблоны, 126
Myghty, механизм шаблонов, 337
MySQL, база данных, 398

N

NameError, исключение (Python), 65
never_cache, декоратор (Django), 349
Nginx, веб-сервер, 394
None, значение (Python), 40
not in, оператор (Python), 45
NullBooleanField, класс (Django), 139

O

OneToOneField, класс (Django), 144
open, функция (Python), 66
operator, модуль (Python), 162
Oracle, база данных, 400
order_by, метод (Django), 159
ORM (Object-Relational Mapper – объектно-реляционное отображение), 122, 135
  преимущества, 135
  специализированные менеджеры (Django), 323

P

Pastebin, пример приложения, 290
  models.py, файл, 291
  ограничение числа отображаемых записей, 300
  определение адресов URL, 294
  очистка с помощью задания cron, 302
  подсветка синтаксиса, 301
  создание шаблонов, 293
  тестирование, 296
path, переменная (объекты запросов), 179
patterns, функция (Django), 172
PIL (Python Imaging Library), установка (пример приложения фотогалереи), 221
pop, метод (Python), 60
popitem, метод (Python), 60
POST, структура данных (Django), 177
PostgreSQL, база данных, 397
process_request, метод (Django), 181
process_response, метод (Django), 182
process_view, метод (Django), 181
Prototype, библиотека JavaScript, 275
.py, расширение файлов, 37
.pyc, расширение файлов, 37
PyCha, библиотека, 321
.pyo, расширение файлов, 37
Python
  арифметические операторы, 38, 41
  битовые операторы, 42
  версии, 380
  выражения и инструкции, 72
  выражения-генераторы, 48
  выяснение номера версии, 381
  генераторы списков, 47
  дополнения, 386
    Easy Install, 386
    IPython, 386
  загрузка, 380
  извлечение срезов последовательностей, 44
  интерактивная оболочка, 35
  интерактивная оболочка, использование Django, 36
  исследование исходных текстов, 225

Python
классы
вложенные классы, 83
динамические атрибуты экземпляра, 91
и модели Django, 83
создание, 80
создание подклассов, 82
создание экземпляров, 81
комбинированные операторы присваивания, 38
комментарии, 37
логическиеоператоры, 39
модули
calendar, 331
doctest, 357
unittest, 358
загрузка, 86
импортирование, 85
пакеты, 86
неиспользуемые символы, 39
обработка исключений, 63
try-except блок, 63
try-finally блок, 64
в Django, 66
объектно-ориентированное программирование, 79
вложенные классы, 83
определение классов, 80
создание подклассов, 82
создание экземпляров, 81
объекты
динамические атрибуты экземпляра, 91
изменяемость, 87
операторы, 38
числовые операторы, 41
операторы деления, 41
операторы сравнивания, 39
отступы, 39
оформление программного кода, 91
короткие блоки в одной строке, 92
отступы, 91
пробелы и символы табуляции, 92
строки документирования, 92
пакеты, 86, 97
переменные, 38
последовательности, 56
кортежи
с единственным элементом, 85
распаковывание в функциях, 75

Python
регулярные выражения, 84
re, модуль, 84
регулярные выражения в файлах URLconf, 171
словари
структура данных COOKIES, 178
структура данных FILES, 180
структура данных META, 180
структура данных REQUEST, 178
структура данных session, 179
структуры данных GET и POST, 177
строки, сырые строки, 103
типы данных
complex, 41
Decimal, 41
float, 40, 41
int, 40, 41
long, 40
итерируемые, 43
кортежи, 43, 54
логический, 40
последовательности, 43
словари, 57
списки, 43, 46
строки, 43, 49
числовые, 40
управление потоком выполнения, 60
условная инструкция, 61
циклы, 61
условная инструкция, 61
установка
тестирование, 384
установка в Linux, 381
установка в Mac OS X, 381
установка в UNIX, 381
установка в Windows, 381
файлы
инициализация, 225
функции, 67
* и **, 75
вызов, 67
именованные аргументы, 68
декораторы, 73, 78
объявление, 67
аргументы со значениями по умолчанию, 69
представления (Django), 183
функции, как обычные объекты, 70
в Django, 71

ссылки, 70
циклы, 61
Python Extesions for Windows, библиотека, 381
Python-Markdown, модуль, загрузка, 253
python-mode, расширение (Emacs), 410
PYTHONPATH, переменная окружения, 95

Q

Q, в комбинации с именованными аргументами, 161
Q, класс (Django), 161
QueryDict, класс (Django), 177
QuerySet, класс (Django), 154
как запрос к базе данных, 155
как контейнер, 156
как строительный блок, 157
настройка запросов, 162
сортировка результатов запросов, 159

R

range, функция (Python), 56, 57
raw_post_data, переменная (объекты запросов), 180
RCS (Revision Control System – система управления изменениями), 403
re, модуль (Python), 84
read, метод (Python), 67
readlines, метод (Python), 67
render_to_response, функция (Django), 188
replace, метод (Python), 51
repr, функция (Python), 56
REQUEST, структура данных (Django), 178
reverse, метод (Django), 159
reverse, метод (Python), 88
reversed, функция (Python), 47
rindex, метод (Python), 51
rstrip, метод (Python), 51, 67
runserver, команда (Django), 98

S

safe, фильтр (Django), 262
save, метод (Django), 201, 228
SCCS (Source Code Control System – система управления исходными текстами), 403
search, функция (Python), 84
select_related, метод (Django), 160
session, структура данных (Django), 179
setdefault, метод (Python), 59, 60
setInterval, функция (JavaScript), 286
settings.py, файл, 97
настройка базы данных, 102
правка, 100, 106
simple.direct_to_template, универсальное представление (Django), 185
sort, метод (Python), 47
sorted, функция (Python), 47, 57
Sphinx, поисковый механизм, 269
split, метод (Python), 51
splitlines, метод (Python), 51
SQL (Structured Query Language – структурированный язык запросов), 122
sql, команда (Django), 154
sqlall, команда (Django), 154
sqlclear, команда (Django), 154
sqlcustom, команда (Django), 154
sqlindexes, команда (Django), 154
SQLite, база данных, 396
настройка в примере приложения блога, 102
создание таблиц, 104
sqlreset, команда (Django), 154
ssi, тег (Django), 199
startapp, команда (Django), 100
startproject, команда (Django), 96
startswith, метод (Python), 51
stderr, поток, 373
stdin, поток, 374
stdout, поток, 373
str, функция (Python), 56
stringvalue, декоратор (Django), 335
strip, метод (Python), 51
Subversion, система управления версиями, 403
краткий обзор, 404
sum, функция (Python), 57
swapcase, метод (Python), 51
syncdb, команда (Django), 104, 154
Syndication, приложение (Django), 314
настройка лент новостей, 314
определение адреса URL ленты, 316

Syntax Highlighter, утилита (JavaScript), 301
SyntaxError, исключение (Python), 65

T

templatetags, каталог (Django), 328
Terminal, приложение, 368
TextField, класс (Django), 138
TextMate, текстовый редактор, 411
ThumbnailImageField, класс (пример приложения фотогалереи)
  использование, 229
  создание, 224
title, метод (Python), 51
Trac, система управления проектами, 409
True, значение (Python), 39
try-except блок, 63
try-finally блок, 64
tuple, функция (Python), 56
TypeError, исключение (Python), 65

U

UnboundLocalError, исключение (Python), 66
unicode, функция (Python), 56
unittest, модуль (Python), 358
Universal Feed Parser, парсер лент новостей, 317
UNIX
  объяснение имен программ, 370
  установка Python, 381
update, метод (Python), 60
upper, метод (Python), 47, 51
url, функция (Django), 172
URLconf, файл (Django), 171
URLField, класс (Django), 138
urls.py, файл, 97
  правка, 106
user, переменная (объекты запросов), 180

V

ValueError, исключение (Python), 66
values, метод (Django), 160
values, метод (Python), 60
values_list, метод (Django), 160
vCard, генерирование, 319
views.py, файл, 100
Vim, текстовый редактор, 411
vObject, модуль, 319

W

while, цикл (Python), 61
Widget, класс (Django), 212
Windows
  использование командной строки, 367
  установка Python, 381
write, метод (Python), 67
writelines, метод (Python), 67
WSGI (Web Server Gateway Interface – шлюзовой интерфейс веб-сервера), 394

X

XHTML, язык разметки, 122

Y

Yahoo! User Interface (YUI) , библиотека JavaScript, 275
YAML, формат файлов, 417

Z

zip, функция (Python), 57

А

абстрактные базовые классы (Django), 146
адреса URL (Django), 170
  include, функция, 174
  url, функция, 172
  URLconf, файл, 171
  вызываемые представления, 175
  модель запрос-ответ HTTP, 176
    объекты запросов, 177
    объекты ответов, 180
    промежуточная обработка, 181
  несколько объектов patterns, 173
альтернативные системы шаблонов, 336
анонимные функции (Python), 72
  lambda, ключевое слово, 73
  в Django, 73
аргументы команд, 369, 371
аргументы со значениями по умолчанию в функциях (Python), 69
арифметические операторы (Python), 38, 41

атрибуты (Django)
вычисляемые атрибуты, 136
добавление в поля, 227
объектов запросов, 179

Б

базовые классы, 80
базовые шаблоны
расширение, 310
базы данных, 121
Django
использование возможностей SQL, не предоставляемых платформой Django, 164
синхронизация, 153
создание и изменение, 152
IBM DB2, 400
Microsoft SQL Server, 400
MySQL, 398
Oracle, 400
PostgreSQL, 397
SQLite, 102, 396
настройка, 101
создание таблиц, 104
поддержка, 400
безопасность моделей (Django), 137
библиотеки тегов (Django), 328
битовые операторы (Python), 42
блоги
агрегатор новостей Django, 429
блочные теги (шаблоны Django), 196
быстрая разработка в Django, 130

В

ввод в командной строке, 374
веб-разработка
MVC (Model-View-Controller – модель-представление-контроллер), 123
контроллеры, 124
методы представлений, 123
способы взаимодействия, 121
способы хранения данных, 121
веб-сайты
поиск приложений Django, 413
регистрация разработчиков Django, 429
веб-серверы, 389
FastCGI, 395
WSGI (Web Server Gateway Interface – шлюзовой интерфейс веб-сервера), 394
встроенный сервер разработки, 390
запуск сервера разработки, 98
обслуживание статических файлов, 393
подключение Django, 391
установка, 390
ветви (системы управления версиями), 403
виджеты форм (Django), 211
по умолчанию, 212
включение приложения Flatpages, 244
включение шаблонов (Django), 198
внешние ключи (Django), 140
вспомогательные сценарии (Django), 339
задания cron, 340
встраивание
приложений Django, 414
встроенные функции (Python), 42
встроенный сервер разработки, 390
выбор библиотеки Ajax (пример приложения liveblog), 274
вывод
состояния окружения, 375
вывод в командной строке, 373
вывод содержимого каталогов, 368
выгрузка файлов, подготовка (пример приложения фотогалереи), 219
вызов функций (Python), 67
* и **, 75
именованные аргументы, 68
вызываемые представления (Django), 175
выражения (Python) и инструкции (Python), 72
выражения-генераторы (Python), 48
выразительность моделей (Django), 137
вычисляемые атрибуты, 136

Г

генераторы списков (Python), 47
гибкость Django, 129
глубокое копирование (Python), 90
Горбачев Алекс (Alex Gorbatchev), 301

Д

декораторы, 73
Django
cache_control, 349
cache_page, 348
never_cache, 349
stringvalue, 335
Python
аутентификации для приложения администрирования, 312
функций, 78
деление с усечением дробной части (Python), 41
деления операторы (Python), 41
децентрализованное управление версиями, 405
диаграммы и графики, вывод, 321
динамические адреса URL (Django), 172
динамические атрибуты экземпляра (Python), 82
динамическое содержимое, 122
документация (Django), 429
дополнения для Python, 386
Easy Install, 386
IPython, 386
дублирование последовательностей (Python), 45

Ж

живой блог, определение, 271

З

заголовки HTTP, имеющие отношение к кэшированию, 348
загружаемые файлы, генерирование, 317
vCard, 319
вывод диаграмм и графиков, 321
значения, разделенные запятыми (CSV), 320
конфигурационные файлы Nagios, 318
загрузка
Django
версия в разработке, 388
официальные выпуски, 388
Google App Engine Helper for Django, 419
Google App Engine SDK, 419
Python, 380
Python-Markdown, модуль, 253
загрузка модулей (Python), 86
задания cron
в примере приложения Pastebin, 302
вспомогательные сценарии, 340
заполнение форм (Django), 205
запросы (Django), динамическое создание, 77
запросы (веб-разработка), 121
запуск сервера разработки, 98
значения, разделенные запятыми (CSV), генерирование, 320

И

извлечение срезов, 44
изменение множества объектов, возвращаемых по умолчанию, 324
изменения (системы управления версиями)
запись, 407
передача, 407
слияние, 404
изменяемость объектов (Python), 87
и вызовы методов, 87
имена программ UNIX, 370
именованные аргументы в функциях (Python), 68
именованные группы, 172
именованные и позиционные аргументы, 173
импорт/экспорт данных, вспомогательные сценарии, 341
импортирование
модулей (Python), 85
инициализация
объектов (Python), 90
файлов (Python), 225
инкапсуляция методов (Django), 136
интегрирование App Engine Helper в приложения Django, 421
интегрирование App Engine в приложения Django, 420
интерактивная оболочка (Python), 35
использование Django, 36
использование функции представления в JavaScript, 286
исходные тексты (Python), исследование, 225
итерируемые типы данных (Python), 43

К

каналы (командная строка), 373
каналы IRC (Django), 429
каскады (SQL), поддержка, 166
каталоги
site-packages, установка приложений, 415
вывод содержимого, 368
для разделяемых приложений, создание, 414
подготовка к выгрузке файлов (пример приложения фотогалереи), 219
удаление, 370
каталоги bin, 379
классы (Django)
AutoField, 140
BooleanField, 139
CharField, 138
EmailField, 138
Field, 138
FileField, 139
FilePathField, 139
ForeignKey, 140
Form, 199
Http404, 188
HttpRequest, 177
HttpResponse, 180
ImageField, 227
ImageField, проверка, 222
ImageFieldFile, 227
IPAddressField, 138
Manager, 154
ManyToManyField, 142
Meta, 149
ModelChoiceField, 204
ModelForm, 200
NullBooleanField, 139
OneToOneField, 144
Q, 161
QueryDict, 177
QuerySet, 154
как запрос к базе данных, 155
как контейнер, 156
как строительный блок, 157
настройка запросов, 162
сортировка результатов запросов, 159
TextField, 138
URLField, 138
классы (Django)
Widget, 212
абстрактные базовые классы, 146
классы (Python)
вложенные, 83
динамические атрибуты экземпляра, 91
и модели Django, 83
создание, 80
подклассов, 82
экземпляров, 81
ключи команд, 369, 371
командная строка
ввод и вывод, 373
имена программ UNIX, 370
использование в Mac OS X, 368
использование в Windows, 367
ключи и аргументы, 371
краткий обзор, 368
переменные окружения, 375
пути, 377
команды
Django
dumpdata, 154
loaddata, 154
manage.py shell, 36
runserver, 98
sql, 154
sqlall, 154
sqlclear, 154
sqlcustom, 154
sqlindexes, 154
sqlreset, 154
startapp, 100
startproject, 96
syncdb, 104, 154
Python
del, 59
ключи и аргументы, 369, 371
комбинированные операторы присваивания (Python), 38
комментарии (Python), 37
комментарии пользователей, поддержка в Django, 269
конкатенация последовательностей (Python), 45
контактная информация (vCard, генерирование), 319
контекст шаблонов (Django), 192
контроллеры (в архитектуре MVC), 124
контроллеры (в веб-разработке), 124

конфигурационные файлы (Nagios), генерирование, 318
конфликт ключей, 59
копирование
объектов и изменяемость, 89
последовательностей (Python), 45
программного кода App Engine в проект, 420
репозиториев (система управления версиями), 408
корневые адреса URL, регулярные выражения, 171
кортежи (Python), 43, 54
с единственным элементом, 85
кэширование (Django), 296, 343
добавление промежуточной обработки, 345
кэширование всего сайта, 347
объектов, 350
определение производительности сервера, 344
стратегии, 347
cache, тег шаблонов, 351
кэширование отдельных представлений, 348
управление заголовками, имеющими отношение к кэшированию, 348
тестирование, 346
типы механизмов кэширования, 352
db, 353
dummy, 353
file, 353
locmem, 353
memcached, 353
установка типа кэширования, 345

Л

ленты новостей
Universal Feed Parser, парсер, 317
логические операторы (Python), 39
логический тип данных (Python), 40
ложный срез (Python), 45

М

менеджеры (Django)
методы
именование, 325
создание, 325
методы (Django)
delete, 229
порядок выполнения операций, 229
distinct, 159
extra, 162
is_valid, 208
order_by, 159
process_request, 181
process_response, 182
process_view, 181
reverse, 159
save, 201, 228
select_related, 160
values, 160
values_list, 160
инкапсуляция, 136
методы (Python)
для строк, 51
для файлов, 66
и изменяемость, 87
словарей, 60
списков, 47
методы представлений (в веб-разработке), 123
многотабличное наследование (Django), 148
модели
определение
Pastebin, пример приложения, 291
модели (Django)
Meta, класс, 149
ORM (Object-Relational Mapper – объектно-реляционное отображение), 135
абстрактные базовые классы, 146
базы данных
использование возможностей SQL, не предоставляемых платформой Django, 164
создание и изменение, 152
безопасность, 137
внешние ключи, 140
выразительность, 137
инкапсуляция методов, 136
многотабличное наследование, 148
наследование, 145
ограничение отношений, 145
определение, 135
отношения между моделями, 140

модели (Django)
параметры модели в приложение администрирования, 151
первичные ключи, 139
переносимость, 136
регистрация, 106
система управления содержимым, пример приложения, 248
создание и изменение баз данных, 152
типы полей, 137
уникальные значения, 139
фотогалерея, пример приложения, 218
модели (в архитектуре MVC), 123
модификация
форм, основанных на моделях (Django), 203
модули (Python)
calendar, 331
django.shortcuts, 188
operator, 162
Python-Markdown, загрузка, 253
re, 84
загрузка, 86
импортирование, 85
пакеты, 86

Н

наследование (Django), 145
абстрактные базовые классы, 146
многотабличное наследование, 148
шаблоны, 198
настройка
базовых шаблонов, расширение, 310
приложения администрирования (Django), 307
расширение базовых шаблонов, 310
с помощью декораторов аутентификации, 312
с помощью параметра fieldsets, 308
создание представлений, 312
приложения на платформе App Engine, 417
программного кода платформы Django, 343
универсальных представлений (Django) с помощью словарей, 295
настройка и тестирование библиотеки Ajax (пример приложения liveblog), 282
несвязанные формы (Django), 205
нормализация данных формы, 209

О

обработка исключений
Django, 66
Python, 63
try-except блок, 63
try-finally блок, 64
в Django, 66
Http404, класс, 188
обработка ошибок в приложении Flatpages, 245
объектно-ориентированное программирование (Python), 79
вложенные классы, 83
модели Django, 83
определение классов, 80
создание подклассов, 82
создание экземпляров, 81
объекты (Python)
изменяемость, 87
и вызовы методов, 87
объекты запросов (Django), 177
объекты ответов (Django), 180
объявление функций (Python), 67
* и **, 75
аргументы со значениями по умолчанию, 69
ограничения между моделями
ограничение отношений, 145
окружение (командная строка), 375
операторы
последовательностей, 45
операторы (Python), 38
числовые, 41
определение
модели
система управления содержимым, пример приложения, 248
файлов с начальными запросами на языке SQL, 165
форм (Django), 199
определение адресов URL, в примере приложения Pastebin, 294
определение моделей (Django), 135

оснастки (Django), 105, 166
ответы (веб-разработка), 121
отношение многие-к-многим, 142
отношение многие-к-одному, 141
отношение один-к-одному, 144
отношения между моделями (Django), 140
отображение приложения администрирования, изменение, 308
отображение форм (Django), 209
отображения
словари, 57
отрицательные индексы в последовательностях Python, 44
отступы (Python), 39, 91
официальные выпуски Django, 388
оформление программного кода (Python), 91
короткие блоки в одной строке, 92
отступы, 91
пробелы и символы табуляции, 92
строки документирования, 92
оценка приложений Django, 413

П

пакеты (Python), 86, 97
параметры отображения форм в приложении администрирования (Django), 152
параметры форматирования списков в приложении администрирования (Django), 152
первичные ключи (Django), 139
передача изменений в системы управления версиями, 407
передача собственных приложений, 415
переменное число аргументов (Python), 75
переменные (Python), 38
переменные окружения, 95, 375
переменные окружения, 375
перенаправление (командная строка), 373
переносимость моделей (Django), 136
переопределение виджетов по умолчанию (Django), 212
поверхностное копирование (Python), 45, 89
подготовка к выгрузке файлов (пример приложения фотогалереи), 219
поддержка производственного процесса в примере приложения системы управления содержимым, 268
подсветка синтаксиса, в примере приложения Pastebin, 301
позиционные и именованные аргументы, 173
поиск приложений Django, 413
получение значений переменных окружения, 376
поля (Django), добавление атрибутов, 227
порядок выполнения операций в методе delete (Django), 229
последовательности (Python), 43
извлечение срезов, 44
кортежи, 43, 54
с единственным элементом, 85
распаковывание в функциях, 75
списки, 43, 46
строки, 43, 49
постраничный просмотр, поддержка в Django, 269
права доступа, 314
к данным, 103
правка
INSTALLED_APPS, кортеж, 100
settings.py, файл, 100, 106
urls.py, файл, 106
представления (Django), 183
вызываемые представления, 175
как функции на языке Python, 183
собственные представления, 188
универсальные представления, 184, 260, 290
представления (SQL), поддержка, 165
представления (в архитектуре MVC), 123
представления административного раздела в примере приложения системы управления содержимым, 257
преобразование
разметки Markdown в разметку HTML, 253
приглашение к вводу (командная строка), 368
приложение администрирования
параметры модели, 151
регистрация, 108

приложение администрирования (Django), 307
настройка, 105, 307
расширение базовых шаблонов, 310
с помощью декораторов аутентификации, 312
с помощью параметра fieldsets, 308
создание представлений, 312
приложения (Django)
Flatpages, 243
включение, 244
обработка ошибок, 245
создание шаблонов, 245
тестирование, 246
liveblog, пример приложения
XML или JSON, 280
выбор библиотеки Ajax, 274
использование функции представления в JavaScript, 286
настройка и тестирование библиотеки Ajax, 282
создание функции представления, 284
структура каталогов, 275
установка библиотеки Ajax, 281
шаблоны, 276
интегрирование App Engine, 420
интегрирование App Engine Helper, 421
оценка, 413
передача, 415
перенос на платформу App Engine, 422
добавление данных, 424
доступ к объектам App Engine, 423
копирование программного кода App Engine, 420
опробование, 423
поиск, 413
проектирование, 273
система управления содержимым, пример приложения, 247
urls.py, файл, 255
views.py, файл, 260
возможные улучшения, 268
импортрование, 250
использование формата Markdown, 252
приложения (Django)
система управления содержимым, пример приложения
отображение статей, 263
поддержка производственного процесса, 268
поиск, 264
представления административного раздела, 257
создание модели, 248
специализированные менеджеры, 251
управление пользователями, 267
шаблоны, 261
создание
на платформе App Engine, 425
тестирование, 356
doctest, модуль (Python), 357
unittest, модуль (Python), 358
запуск тестов, 358
инструменты, 361
моделей, 359
установка, 414
фотогалерея, пример приложения, 217
models.py, файл, 218
PIL (Python Imaging Library), установка, 221
ThumbnailImageField, класс, создание, 224
настройка адресов URL, 231
подготовка к выгрузке файлов, 219
проверка поля ImageField, 222
шаблоны, 235
приложения на платформе Django
blog, пример проекта
приложение администрирования, 105
регистрация, 108
создание проекта, 96
создание шаблона, 112
создание шаблона URL, 114
шаблоны
базовый шаблон, 115
фильтры, 118
django-admin.py, утилита, 96
запуск сервера разработки, 98
модели
создание, 101

приложения на платформе Django
настройка базы данных, 101
создание приложения, 100
теги переменных, 112
функции представления, 113
примеры приложений
liveblog, 273
XML или JSON, 280
выбор библиотеки Ajax, 274
использование функции представления в JavaScript, 286
настройка и тестирование библиотеки Ajax, 282
проектирование, 273
создание функции представления, 284
структура каталогов, 275
установка библиотеки Ajax, 281
шаблоны, 276
Pastebin, 290
models.py, файл, 291
ограничение числа отображаемых записей, 300
определение адресов URL, 294
очистка с помощью задания cron, 302
подсветка синтаксиса, 301
создание шаблонов, 293
тестирование, 296
запуск сервера разработки, 98
модели
создание, 101
настройка базы данных, 101
приложение администрирования, 105
система управления содержимым, 247
urls.py, файл, 255
views.py, файл, 260
возможные улучшения, 268
импортирование, 250
использование формата Markdown, 252
отображение статей, 263
поддержка производственного процесса, 268
поиск, 264
представления административного раздела, 257
создание модели, 248
примеры приложений
система управления содержимым
специализированные менеджеры, 251
управление пользователями, 267
шаблоны, 261
создание приложения, 100
создание проекта, 96
фотогалерея, 217
PIL (Python Imaging Library), установка, 221
ThumbnailImageField, класс, создание, 224
настройка адресов URL, 231
подготовка к выгрузке файлов, 219
проверка поля ImageField, 222
шаблоны, 235
присваивание значений переменным окружения, 376
проверка форм (Django), 207
программное обеспечение управления проектами, 409
проект (Django), создание, 96
производительность сервера, определение, 344
промежуточная обработка (Django), 181
промежуточная обработка запросов (Django), 182
промежуточная обработка ответов (Django), 182
процессоры контекста (Django), 192
пути
обновление для Python, 382
путь (командная строка), 377

Р

распаковывание последовательностей (Python) в функциях, 75
расположение элементов приложения администрирования, изменение, 308
расширение
базовых шаблонов, 310
системы шаблонов (Django), 326
расширения файлов (Python), 37
регистрация
моделей (Django), 106
разработчиков Django, 429
регистрация в приложении администрирования, 108

регулярные выражения (Python), 84
re, модуль, 84
в файлах URLconf, 171

С

связанные формы (Django), 205
сгруппированные данные, 136
сеансы, 178
символы завершения строк в файлах (Python), 67
символы неиспользуемые (Python), 39
синтаксис языка шаблонов (Django), 193
система управления содержимым
Flatpages, приложение, 243
включение, 244
обработка ошибок, 245
создание шаблонов, 245
тестирование, 246
определение, 242
пример приложения, 247
urls.py, файл, 255
views.py, файл, 260
возможные улучшения, 268
импортрование, 250
использование формата Markdown, 252
отображение статей, 263
поддержка производственного процесса, 268
поиск, 264
представления административного раздела, 257
создание модели, 248
специализированные менеджеры, 251
управление пользователями, 267
шаблоны, 261
система шаблонов (Django), расширение, 326
специализированные теги, 326
специализированные фильтры, 333
теги включения, 330
системы управления версиями, 403
децентрализованное управление версиями, 405
слияние изменений, 404
ствол и ветви, 403
централизованное управление версиями, 404
слабая зависимость Django, 129
слияние изменений (системы управления версиями), 404
словари (Python), 57
в Django, 59
настройка универсальных представлений, 295
в качестве аргументов со значениями по умолчанию, 69
структура данных COOKIES, 178
структура данных FILES, 180
структура данных META, 180
структура данных REQUEST, 178
структура данных session, 179
структуры данных GET и POST, 177
собственные запросы (SQL), поддержка, 168
собственные классы (пример приложения фотогалереи)
использование, 229
создание, 224
собственные представления (Django), 188
собственные типы данных (SQL), поддержка, 166
собственные функции (SQL), поддержка, 166
соглашения об именовании частичных шаблонов (Django), 332
создание экземпляров (Python), 81, 90
сортировка
результатов запросов, 159
списков (Python), 47
сохранение
миниатюр в примере приложения фотогалереи, 228
спам в приложениях Pastebin, 299
специализированные менеджеры (Django), 323
методы
именование, 325
создание, 325
специализированные фильтры (Django), 333
спецификаторы строк (Python), 52
списки (Python), 43, 46
в качестве аргументов со значениями по умолчанию, 69
выражения-генераторы, 48
генераторы списков, 47
сортировка, 47, 88

списки рассылки (Django), 429
способы взаимодействия (в веб-разработке), 121
способы хранения данных (в веб-разработке), 121
справочная система командной строки, 370
сравнивания операторы (Python), 39
ссылки на объекты функций, 70
статические члены класса, 80
статическое содержимое, 122
ствол (системы управления версиями), 403
столбцы (в базе данных), 122
строки
  Юникода, 53
строки (Python), 43, 49
  конкатенация, 45
  спецификаторы, 52
  сырые строки, 103
  форматирование, 53
строки (в базе данных), 122
строки документирования (Python), 92
суперпользователь (Django), создание учетной записи, 104
схемы с собственными начальными запросами на языке SQL, 165
сырые строки (Django), 52
сырые строки (Python), 52, 103
  в регулярных выражениях, 84

Т

таблицы (в базе данных), 122
  создание, 104
тег блока (Django), 113
теги (Django)
  блочные, 198
теги (шаблоны Django), 191, 195
  include, 199
  ssi, 199
теги включения (Django), создание, 330
теги переменных (Django), 112
текстовые редакторы, 410
тестирование
  ImageField, класс (Django), пример приложения фотогалереи, 222
  Pastebin, пример приложения, 296
  инструменты, 361
  приложений (Django), 356
  приложения Flatpages, 246
  программного кода платформы Django, 362
  установки Django, 389
  установки Python, 384
типы данных
  complex (Python), 41
  Decimal (Python), 41
  float (Python), 40, 41
  int (Python), 40, 41
  long (Python), 40
  логический (Python), 40
  переменных, 38
  числовые (Python), 40
типы данных (Python)
  итерируемые, 43
  последовательности, 43
    извлечение срезов, 44
    конкатенация, 45
    копирование, 45
    кортежи, 43, 54
    операции, 45
    списки, 43, 46
    строки, 43, 49
  словари, 57
  функции
    для последовательностей, 56
типы механизмов кэширования (Django), 352
  db, 353
  dummy, 353
  file, 353
  locmem, 353
  memcached, 353
типы полей (Django), 137
триггеры (SQL), поддержка, 166
тройные кавычки в строках Python, 53

У

удаление
  каталогов, 370
  миниатюр в примере приложения фотогалереи, 228
  устаревших записей (в примере приложения Pastebin), 302
  файлов, 368
универсальные представления (Django), 184, 290
  настройка с помощью словарей, 295
  пример приложения системы управления содержимым, 260

уникальные значения (Django), 139
управление пользователями в примере приложения системы управления содержимым, 267
управление потоком выполнения (Python), 60
условная инструкция, 61
циклы, 61
условная инструкция (Python), 61
установка
Django, 388
приложений, 414
тестирование, 389
Python
в Mac OS X, 381
в UNIX/Linux, 381
в Windows, 381
тестирование, 384
веб-сервера Apache и модуля mod_python, 390
установка библиотеки Ajax (пример приложения liveblog), 281
участие в проекте Django, 428

Ф

фабричные функции (Python), 42
файлы
с начальными запросами на языке SQL, 165
файлы (Python), 66
инициализация, 225
методы, 66
удаление, 368
фильтры (Django)
safe, 262
slice, 300
специализированные фильтры, 333
фильтры (шаблоны Django), 191, 194
форматирование
строк, 53
формы (Django), 199
виджеты, 211
по умолчанию, 212
заполнение, 205
несвязанные, 205
нормализация данных, 209
определение, 199
основанные на моделях, 200
модификация, 203
сохранение, 201
формы (Django), 199
отображение, 209
проверка, 207
связанные, 205
создание подклассов форм, 204
фотогалерея, пример приложения, 217
PIL (Python Imaging Library), установка, 221
ThumbnailImageField, класс, создание, 224
настройка адресов URL, 231
подготовка к выгрузке файлов, 219
проверка поля ImageField, 222
шаблоны, 235
функции
представления (Django), 183
функции (Django)
get_list_or_404, 188
get_object_or_404, 188
include, 174
render_to_response, 188
функции (Python), 67
* и **, 75
all, 56
any, 56, 57
bool, 40
enumerate, 56, 62
в моделях платформы Django, 62
eval, 56
group, 84
groups, 84
help, 93
len, 56, 57
list, 56
match, 85
max, 56, 57
min, 56
open, 66
range, 56, 57
repr, 56
reversed, 47
search, 84
sorted, 47, 57
str, 56
sum, 57
tuple, 56
unicode, 56
zip, 57
анонимные функции, 72
lambda, ключевое слово, 73
в Django, 73

функции (Python)
вызов, 67
именованные аргументы, 68
декораторы, 73, 78
для последовательностей, 56
объявление, 67
аргументы со значениями по умолчанию, 69
числовые функции, 42
функции Python, как обычные объекты, 70
в Django, 71
ссылки, 70
функции представления (Django), 113
функция поиска в примере приложения системы управления содержимым, 264

Ц

централизованное управление версиями, 404
циклы (Python), 61

Ч

частичные шаблоны (Django), именование, 332
числовые операторы (Python), 41
числовые типы данных (Python), 40
числовые функции (Python), 42

Ш

шаблоны
синтаксис языка шаблонов, 193
шаблоны (Django), 112, 191
liveblog, пример приложения, 276
базовый шаблон, 115
блочные теги, 196
внедрение операций JavaScript, 283
для приложения Flatpages, 245
контекст, 192
наследование, 198
пример приложения системы управления содержимым, 261
расширение, 326
создание, 112
Pastebin, пример приложения, 293
специализированные теги, 326
специализированные фильтры, 333
шаблоны (Django)
теги, 191, 195
include, 199
ssi, 199
наследование, 198
теги включения, 330
фильтры, 118, 191, 194
фотогалерея, пример приложения, 235
шаблоны (Django) базовые шаблоны расширение, 310
шаблоны (в архитектуре MVC, 126
шаблоны URL (Django), 114
создание, 114

Э

экранирование разметки HTML, 262

Я

язык шаблонов, 122

Мэтью РАССЕЛЛ

Dojo. Подробное руководство

560 стр., книга в продаже

С помощью Dojo, высоконадежного инструментария JavaScript, вы сможете быстрее и проще создавать веб-приложения и сайты, основанные на применении JavaScript или технологии Ajax. Издание представляет собой наиболее полный сборник документации по Dojo, снабженный развернутыми комментариями.

Демонстрируются эффективные приемы работы с обширным набором утилит, реализация различных пользовательских механизмов, методы воспроизведения анимационных эффектов. Рассматриваются проекты, входящие в состав библиотеки DojoX, с инструментами сборки и платформой модульного тестирования.

Занимаетесь ли вы разработкой веб-приложений с применением DHTML в одиночку или в составе команды разработчиков, эта книга поможет вам эффективно воплотить новые идеи, значительно обогатив интерфейс веб-приложений.

Дэвид ФЛЭНАГАН

JavaScript. Подробное руководство, 5-е издание

992 стр., книга в продаже

Пятое издание книги полностью обновлено. Рассматриваются взаимодействие с протоколом HTTP и применение технологии Ajax, обработка XML-документов, создание графики на стороне клиента с помощью тега <canvas>, пространства имен в JavaScript, необходимые для разработки сложных программ, классы, замыкания, Flash и встраивание сценариев JavaScript в Java-приложения.

Часть I знакомит с основами JavaScript. В части II описывается среда разработки сценариев, предоставляемая веб-броузерами. Многочисленные примеры демонстрируют, как генерировать оглавление HTML-документа, отображать анимированные изображения DHTML, автоматизировать проверку заполнения форм, создавать всплывающие подсказки с использованием Ajax, как применять XPath и XSLT для обработки XML-документов, загруженных с помощью Ajax. Часть III – обширный справочник по основам JavaScript (классы, объекты, конструкторы, методы, функции, свойства и константы, определенные в JavaScript 1.5 и ECMAScript 3). Часть IV – справочник по клиентскому JavaScript. Описываются API веб-броузеров, стандарт DOM API Level 2 и недавно появившиеся стандарты: объект XMLHttpRequest и тег <canvas>.

Бер БИБО, Иегуда КАЦ

jQuery
Подробное руководство по продвинутому JavaScript

384 стр., книга в продаже

Издание представляет собой введение и справочное руководство по jQuery – мощной платформе для разработки веб-приложений. Подробно описывается, как выполнять обход документов HTML, обрабатывать события, добавлять поддержку технологии Ajax в свои веб-страницы, воспроизводить анимацию и визуальные эффекты. Уникальные «лабораторные страницы» помогут закрепить изучение каждой новой концепции на практических примерах. Рассмотрены вопросы взаимодействия jQuery с другими инструментами и платформами и методы создания модулей расширения для этой библиотеки.

Книга предназначена для разработчиков, знакомых с языком JavaScript и технологией Ajax и стремящихся создавать краткий и понятный программный код. Уникальная способность jQuery составлять «цепочки» из команд позволяет выполнять несколько последовательных операций над элементами страницы, в результате чего код сокращается втрое.

Николас ЗАКАС, Джереми МАК-ПИК, Джо ФОСЕТ

AJAX для профессионалов

488 стр., книга в продаже

AJAX объединяет такие проверенные временем технологии, как CSS, XML и JavaScript, и предоставляет разработчикам возможность создавать более сложные и более динамичные пользовательские интерфейсы, а также отменяет господство стандарта «щелкни и жди», начавшееся еще со времен появления Всемирной паутины.

В книге рассматриваются различные способы выполнения запросов и поясняется, когда и при каких обстоятельствах должна использоваться та или иная методика. Из книги вы узнаете о различных методиках и шаблонах проектирования AJAX, применяемых для организации взаимодействия между клиентом и сервером в ваших веб-приложениях, на вашем сайте. Материал последующих глав основывается на предыдущих, поэтому к концу книги вы получите практические знания, необходимые для реализации ваших собственных решений на основе AJAX.

Книга предназначена разработчикам, желающим повысить эргономические качества своих сайтов и веб-приложений. Знакомство с JavaScript, HTML и CSS обязательно, как и некоторый опыт работы с технологиями разработки серверных сценариев, таких как PHP и .NET.

Марк САММЕРФИЛД

Программирование на Python. Подробное руководство

608 стр., книга в продаже

Третья версия языка Python сделала его еще более мощным, удобным, логичным и выразительным. Книга «Программирование на Python 3» написана одним из ведущих специалистов по этому языку, обладающим многолетним опытом работы с ним. Издание содержит все необходимое для практического освоения языка: написания любых программ с использованием как стандартной библиотеки, так и сторонних библиотек для языка Python 3, а также создания собственных библиотечных модулей.

Автор начинает с описания ключевых элементов Python, знание которых необходимо в качестве базы. Затем обсуждаются более сложные темы, поданные так, чтобы читатель мог постепенно наращивать свой опыт: распределение вычислительной нагрузки между несколькими процессами и потоками, использование сложных типов данных, управляющих структур и функций, создание приложений для работы с базами данных SQL и с файлами DBM.

Книга может служить как учебником, так и справочником. Текст сопровождается многочисленными примерами, доступными на специальном сайте издания. Весь код примеров был протестирован с окончательным релизом Python 3 в ОС Windows, Linux и Mac OS X.

Джеффри ФРИДЛ

Регулярные выражения, 3-е издание

608 стр., книга в продаже

Эта книга откроет перед вами секрет высокой производительности. Тщательно продуманные регулярные выражения помогут избежать долгих часов утомительной работы и решить свои проблемы за 15 секунд. Ставшие стандартной возможностью во многих языках программирования и популярных программных продуктах, включая Perl, PHP, Java, Python, Ruby, MySQL, VB.NET, C# (и других языках платформы .NET), регулярные выражения позволят вам автоматизировать сложную и тонкую обработку текста.

Написанное простым и доступным языком, это издание позволит программистам легко разобраться в столь сложной теме. Рассматривается принцип действия механизма регулярных выражений, сравниваются функциональные возможности различных языков программирования и инструментальных средств, подробно обсуждается оптимизация, которая дает основную экономию времени! Вы научитесь правильно конструировать регулярные выражения для самых разных ситуаций, а большое число сложных примеров даст возможность сразу же использовать предлагаемые ответы для выработки элегантных и экономичных практических решений широкого круга проблем.

Марк ЛУТЦ

Изучаем Python, 3-е издание

848 стр., книга в продаже

Третье издание бестселлера позволит быстро и эффективно овладеть базовыми основами языка Python, который идеально подходит для разработки самостоятельных программ и сценариев. Если вы хотите понять, почему Python выбирают такие компании, как Google и Intel, Cisco и Hewlett-Packard, то эта книга станет для вас лучшей отправной точкой. Она основана на материалах учебных курсов, которые автор ведет уже на протяжении 10 лет.

Представлены основные типы объектов в языке Python, порядок их создания и работы с ними, а также функции как основной процедурный элемент языка. Рассматриваются методы работы с модулями и классами. Включены описания моделей и инструкций обработки исключений, а также обзор инструментов разработки, используемых при создании крупных программ. Обсуждаются изменения в версии 3.0.

Ноа ГИФТ, Джереми ДЖОНС

Python в системном администрировании UNIX и Linux

512 стр., книга в продаже

Книга сисадмина с 10-летним стажем и инженера-программиста демонстрирует, как можно с помощью языка Python эффективно решать разнообразные задачи управления серверами UNIX и Linux. Каждая глава посвящена определенной задаче, например многозадачности, резервному копированию данных или созданию собственных инструментов командной строки, и предлагает практические методы ее решения на языке Python.

Среди рассматриваемых тем: организация ветвления процессов и передача информации между ними с использованием сетевых механизмов, создание интерактивных утилит с графическим интерфейсом, организация взаимодействия с базами данных и создание приложений для Google App Engine. Авторами создана виртуальная машина на базе Ubuntu, которая включает исходные тексты примеров из книги.

Джеффри Форсье – системный администратор и веб-разработчик компании Digital Pulp Inc. (Нью-Йорк), занимающейся интерактивным рекламным бизнесом и разработкой веб-приложений.

Пол Биссекс занимается разработкой платформы Django с первых дней ее появления. Создатель и администратор сайта сообщества Django, инструктор и разработчик приложений в Институте фотографии.

Уэсли Чан – консультант, специализирующийся на техническом обучении и разработке программного обеспечения, автор бестселлера «Core Python Programming» (*corepython.com*).

Категория: Python

Уровень подготовки читателей: средний

Addison-Wesley

www.symbol.ru

Издательство «Символ-Плюс»
(812) 324-5353, (495) 945-8100

С помощью простой и надежной платформы Django, основанной на языке Python, можно создавать мощные веб-решения, написав всего несколько строк программного кода. Ее использование позволяет обеспечить высокую скорость и гибкость разработки, а также решать широкий спектр прикладных задач.

В этой книге три опытных разработчика описывают все приемы, инструменты и концепции, которые необходимо знать, чтобы оптимально использовать Django 1.0, включая все основные особенности новой версии. Авторы раскрывают секреты платформы, давая подробные разъяснения и предоставляя большое количество примеров программного кода, сопровождая их построчным описанием и иллюстрациями.

Руководство начинается с введения в язык Python и основные понятия разработки веб-приложений, затем переходит к обсуждению платформы Django, где подробно рассматриваются основные ее компоненты (модели, представления, шаблоны) и порядок организации взаимодействий между ними.

В процессе чтения книги читателю предлагается создавать свои первые приложения на платформе Django:

- Простой веб-лог (блог)
- Фотогалерея
- Простая система управления содержимым
- «Живой» блог с использованием технологии Ajax
- Инструмент публикации фрагментов программного кода с подсветкой синтаксиса
- Примеры приложений Django, опирающихся на платформу Google App Engine

Далее книга переходит к обсуждению более сложных тем, таких как расширение системы шаблонов, синдицирование, настройка приложения администрирования и тестирование. В приложениях дается ценная информация об использовании командной строки, установке и настройке платформы, инструментах разработки, поиске и оценке существующих приложений на Django, платформе Google App Engine и о том, как принять участие в жизни сообщества пользователей Django.

Да здравствует коммунизм!

ISBN 978-5-93286-167-7