ЧАСТЬ I

Web-приложения. Язык Python. Библиотека Django

Глава 1. Введение в серверное Web-программирование

Глава 2. Язык программирования Python

Глава 3. Библиотека Django

Глава 4. Создание проекта и приложения Django

ГЛАВА 1

Введение в серверное Web-программирование

Не откладывая дела в долгий ящик, сразу же приступим к рассмотрению основных принципов серверного Web-программирования. Мы узнаем, что такое собственно серверные Web-приложения, как они работают, что есть базы данных, таблицы, поля, записи, индексы, модели, контроллеры и шаблоны. Без всего этого мы просто не поймем, о чем пойдет речь в следующих главах книги.

А начнем мы с того, что разберемся, в чем принципиальная разница между статичными Web-страницами и Web-приложениями, и рассмотрим их преимущества и недостатки.

Статичные Web-страницы и Web-приложения — две эпохи в развитии Интернета

В более чем тридцатилетней истории Интернета можно выделить две эпохи. Эпоха первая, давно прошедшая, представляла собой царство статичных Web-страниц — тех самых, что пишутся на языке *HTML* (HyperText Markup Language, язык гипертекстовой разметки) и хранятся в обычных текстовых файлах с расширениями htm или html. Эпоха вторая, которая продолжается и сейчас, ознаменована господством серверных Web-приложений, особых программ, которые получают данные из базы, обрабатывают их и генерируют на основе результатов обработки Web-страницы.

Но почему произошел такой резкий переход от статики к, можно сказать, динамике? Чем Web-дизайнеров не устраивали старые добрые Web-страницы?

Статичные Web-страницы

Написать статичную Web-страницу (для краткости их называют просто Web-страницами) — пара пустяков. Необходимый для их создания язык HTML сейчас знают даже школьники, равно как и язык *CSS* (Cascading Style Sheets, каскадные таблицы стилей), на котором в виде каскадных таблиц стилей и описывается их оформление.

WEB-СКРИПТЫ *НА ЯЗЫКЕ JAVASCRIPT*

Существуют также *Web-сценарии*, или *Web-скрипты*, с помощью которых программируется поведение страниц или отдельных их элементов в ответ на действия посетителя или каких-либо событий, происходящих в Web-обозревателе. Они пишутся на языке программирования JavaScript. Однако в этой книге мы их касаться не будем.

А для разработки Web-страниц подойдет любой текстовый редактор — например, Блокнот, поставляемый в составе Windows.

Давайте создадим Web-страницу с вот таким кодом:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Страница 1</title>
  </head>
  <body>
    <h1>Это страница 1</h1>
    <p>Щелкните гиперссылку внизу, чтобы перейти на страницу 2.</p>
    <p><a href="pages/2.html">Перейти</a>.</p>
  </body>
</html>
```

Эта страница очень проста — она включает лишь заголовок, два абзаца и гиперссылку для перехода на вторую страницу, которую мы скоро сделаем.

Сохраним нашу первую Web-страницу в какой-либо папке в кодировке UTF-8 под именем 1.html.

Теперь создадим еще одну страницу. Ее HTML-код будет таким:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Страница 2</title>
  </head>
  <body>
    <h1>Это страница 2</h1>
    <p>Щелкните гиперссылку внизу, чтобы вернуться на страницу 1.</p>
    <p><a href="../1.html">Перейти</a>.</p>
  </body>
</html>
```

Здесь все то же самое, за единственным отличием — гиперссылка ведет на первую страницу.

Создадим в папке, где хранится первая страница, вложенную папку pages и сохраним в этой папке вторую страницу также в кодировке UTF-8, дав ей имя 2.html.

Откроем страницу 1.html в Web-обозревателе. Выглядеть она будет так, как показано на рис. 1.1.

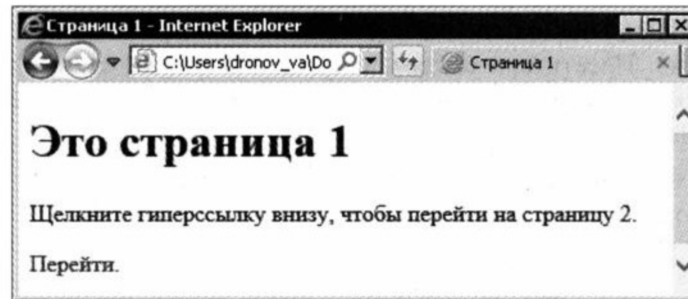

Рис. 1.1. Статичная Web-страница

Щелкнем на гиперссылке, чтобы перейти на вторую страницу, после чего вернемся на первую. Если мы не допустили в HTML-коде ошибок, все должно получиться.

Как видим, сайт на статичных Web-страницах делается очень просто — мы создаем страницы и раскладываем их по папкам. Структура папок отражает структуру самого сайта.

Обрабатываются статичные страницы также очень просто. Когда посетитель набирает в строке ввода адреса Web-обозревателя интернет-адрес нашего сайта и нажимает клавишу <Enter>, Web-обозреватель отправляет компьютеру, располагающемуся в Сети по этому адресу, особый запрос. Операционная система удаленного компьютера передает этот запрос программе *Web-сервера*. Та извлекает из запроса имя файла запрошенной Web-страницы, считывает этот файл и отправляет его Web-обозревателю. Последнему остается лишь получить файл, обработать его и вывести страницу на экран.

Ключевых преимуществ у статичных Web-страниц два. Во-первых, их очень просто создавать — для этого достаточно знать языки HTML и CSS, о которых говорилось ранее. Во-вторых, развернуть на компьютере статичный Web-сайт не составляет ни малейшего труда — нужно лишь установить на него и настроить программу Web-сервера, что можно сделать буквально за пять минут.

Сайты, основанные на статичных Web-страницах, активно создаются до сих пор. Это всевозможные домашние страницы, рекламные сайты и сайты-визитки.

Web-приложения

А теперь представим себе такую ситуацию. Мы создали основанный на статичных страницах корпоративный Web-сайт некоей фирмы, содержащий в своем составе каталог товаров. Какое-то время этот сайт работал и благополучно привлекал клиентов, пока начальство не поставило нам следующие задачи:

1. Реализовать в каталоге фильтрацию товаров по ключевым словам и сортировку их по наименованиям и цене.
2. Создать гостевую книгу.
3. Создать блог для сотрудников фирмы, где они смогли бы публиковать тематические статьи.

Конечно, соорудить некое подобие гостевой книги и блога на статичных Web-страницах можно. Посетители сайта и сотрудники фирмы станут присылать нам свои сообщения и статьи по электронной почте, а мы будем вставлять их в код Web-страниц. Неудобно, но вполне реализуемо.

Но как сделать фильтрацию и сортировку товаров в каталоге? Средствами HTML и CSS это реализовать всяко не получится.

Выход здесь один — создать некую программу, которая будет:

- работать совместно с Web-сервером;
- перехватывать запросы, получаемые Web-сервером, и активизироваться лишь при обращении к интернет-адресу списка товаров;
- извлекать из запросов введенные посетителем ключевые слова для поиска и критерий сортировки (такие данные обычно отправляют как часть интернет-адреса — методом GET);
- считывать из базы данных список товаров, сортировать его согласно заданным критериям и отфильтровывать лишь те товары, чьи наименования включают указанное ключевое слово;
- формировать на основе результирующего списка товаров обычную Web-страницу и передавать ее Web-серверу, который, в свою очередь, отправит ее посетителю.

В этом случае Web-страницы нашего сайта не будут храниться в файлах на жестких дисках компьютера, а станут генерироваться на «лету» особым приложением. Сразу видно, что такой подход решит огромное количество проблем, более того, мы, немного поразмыслив, без труда сможем создать на сайте, в том числе, и гостевую книгу с блогом, написав соответствующие программы.

Такие приложения, работающие совместно с Web-сервером и генерирующие страницы на основе данных, хранящихся в базе и введенных посетителем, носят название *Web-приложений*. Точнее, *серверных Web-приложений*, поскольку они функционируют на стороне Web-сервера.

Клиентские Web-приложения

Существуют также *клиентские Web-приложения*, представляющие собой сложные Web-сценарии и обрабатывающие данные, как полученные от серверного Web-приложения, так и введенные посетителем, на стороне клиента — Web-обозревателя. Но, поскольку в этой книге не рассматривается клиентское Web-программирование, мы говорить о них не будем.

Как правило, сложные сайты включают в себя сразу несколько серверных приложений. Так, отдельное приложение формирует *главную Web-страницу*, которая выводится при обращении непосредственно по адресу сайта, отдельные же приложения генерируют список товаров, гостевую книгу, блог, фотогалерею и прочие разделы сайта. Обычные страницы, наподобие сведений о фирме и сайте или списка контактов, могут также генерироваться разными приложениями, равно как и создаваться одним «универсальным» приложением — это зависит от реализации.

Как мы уже знаем, Web-сайт, основанный на статичных страницах, представляет собой набор файлов и папок. Получив запрос, Web-сервер просто извлекает из него имя запрошенного файла, после чего считывает его с диска и пересылает Web-обозревателю. Например, получив запрос **http://www.somesite.ru/pages/3.html**, Web-сервер прочтет файл 3.html из папки pages, что находится в папке, где помещается сам сайт.

В Web-сайте, основанном на серверных приложениях, все несколько иначе. Каждое составляющее его приложение привязывается к определенному интернет-адресу, который можно рассматривать как «папку», не существующую в реальности. Web-сервер, получив запрос на обращение к такой «папке», запускает соответствующее ей серверное приложение. Скажем, если обратиться по интернет-адресу **http://www.somesite.ru/goods/**, будет запущено приложение, выводящее список товаров.

Самое интересное, что «структура» таких «папок» совершенно не обязана совпадать со структурой файлов и папок, составляющих сайт. С одной стороны, это может внести некоторую путаницу, но, с другой, позволяет реализовать весьма интересные сценарии работы...

Подробный рассказ об интернет-адресах и привязке их к приложениям еще впереди. А пока что давайте рассмотрим преимущества сайтов, основанных на серверных приложениях.

- Возможность хранить данные отдельно (как правило, в базе данных), или, говоря другими словами, отделить их от обработки и вывода. Это может быть полезно, если эти данные используются не только в сайте, но и где-либо еще, скажем, в бухгалтерском или складском приложении.
- Возможность реализовать обработку данных перед их выводом: фильтровать их, сортировать и объединять в группы. Мы можем даже считать количество позиций в списке, вычислять средние значения для параметров позиций и пр.
- Возможность получить данные от посетителя и сохранить их в базе.
- Повысить безопасность, заблокировав доступ к файлам сайта, не предназначенным для чужих глаз (той же базе данных).

Недостатков у подобных сайтов всего два, и оба не являются критичными:

- для создания серверных приложений необходимо дополнительно изучить язык программирования, на котором будут писаться эти приложения, и платформу их разработки. Что, впрочем, несложно;
- помимо собственно Web-сервера, нам понадобится также установить и настроить программное ядро соответствующей платформы. Сделать это также несложно — весь процесс установки и первоначальной настройки занимает несколько минут.

Платформ для разработки серверных приложений существует несколько. Прежде всего, это PHP (Pretty Home Page, симпатичная домашняя страница) — на данный момент имеющая наибольшую популярность. Приложения для этой платформы создаются на особом языке, который также называется PHP. А чтобы облегчить

труд Web-программиста, написано несколько библиотек, реализующих типовые задачи, которые в противном случае придется решать самому разработчику: Zend Framework, Yii и многие другие.

Мы же будем использовать платформу Django. Эта платформа служит для создания Web-приложений на языке Python (и сама, кстати, написана на этом языке).

Базы данных. Реляционные базы данных

Ранее мы уже говорили, что данные, с которыми работают серверные Web-приложения, хранятся в базах данных. Настала пора узнать, что это такое.

Что такое реляционная база данных?

Итак, *база данных* — это файл (или набор файлов), хранящий структурированную определенным образом информацию. Для обработки этой информации используются особые программы, называемые *системами управления базами данных*, или *СУБД*. Примеры СУБД: MySQL, PostgreSQL, Oracle, Microsoft SQL Server и Microsoft Access.

Базы данных делятся на несколько видов по способу структурирования содержащейся в них информации. Мы будем пользоваться *реляционными базами данных* — они служат для хранения информации, организованной в виде связанных друг с другом таблиц, и в настоящее время имеют наибольшее распространение.

Кстати, все упомянутые ранее СУБД обрабатывают реляционные базы данных.

Что хранит реляционная база данных?

Реляционная база хранит структуры, относящиеся к трем различным типам. Давайте их рассмотрим.

Таблицы, поля и записи

Таблица — это набор структурированных данных. Пример таблицы представлен на рис. 1.2.

Эта таблица содержит список товаров из пяти столбцов: категория товара, его наименование, описание, цена и признак того, имеется ли товар в наличии.

Каждая таблица, хранящаяся в базе данных, должна иметь уникальное в пределах этой базы имя. Это нужно для того, чтобы СУБД (да и мы сами) смогла найти эту таблицу.

Отдельная строка таблицы, содержащая реальные данные, называется *записью*. (Строка заголовка, выделенная на рис. 1.2 черным фоном, записью не является, т. к. содержит не реальные данные, описывающие какую-либо статью, а служебные сведения — заголовки столбцов.) На каждый товар, занесенный в таблицу, отводится одна запись.

category	name	description	price	in_stock
Веники	В-1	Классическая конструкция из экологически чистых материалов	100	*
Веники	В-2М	Усовершенствованная конструкция, выполненная с применением особо прочной синтетики	200	
Веники	В-2000	Современная конструкция, выполненная с применением нанотехнологий	2000	*
Метлы	М-1	Классическая конструкция из экологически чистых материалов	150	*
Метлы	М-2014О	Выпущена в честь Зимней олимпиады в Сочи 2014 года	15000	*

Рис. 1.2. Таблица — список товаров

Отдельная ячейка отдельной строки-записи называется *полем*. Можно сказать, что поле — это порция данных, составляющих запись. А сами данные, помещенные в поле, называются его *значением*.

Каждое поле обязано иметь уникальное в пределах таблицы имя. Имена полей, кстати, и приведены в строке заголовка таблицы на рис. 1.2.

Поле способно хранить данные какого-то одного типа: строки, числа, даты и т. п. *Тип* хранимых в поле данных задается при создании поля (и может быть потом изменен, если был задан ошибочно). СУБД не позволит записать, скажем, дату в поле, предназначенное для хранения строк. Типы данных, поддерживаемые большинством баз данных, приведены в табл. 1.1.

Таблица 1.1. *Типы данных, поддерживаемые большинством форматов баз данных*

Название	**Описание**
Строковый	Текст фиксированной длины, содержащий любые символы: буквы, цифры, знаки препинания, пробелы и пр. Максимальная длина текста, хранимого в таком поле, задается при его создании
Целочисленный	Целые числа
С плавающей точкой	Дробные числа
Логический	Значения вида «истина» (true) или «ложь» (false)
Дата	Значения даты
Дата и время	Объединенное значение даты и времени
Memo	Текст произвольной длины, содержащий любые символы: буквы, цифры, знаки препинания, пробелы и пр. Длина хранимого в таком поле текста не ограничена (по крайней мере, очень велика)
Счетчик	Постепенно увеличивающиеся и уникальные в пределах таблицы целые числа. Поля такого типа используются для специальных целей, в частности, в качестве ключевого поля (см. далее)

Несколько разновидностей типов данных

На самом деле, существует несколько разновидностей целочисленного типа данных и типа с плавающей точкой, различающихся величинами чисел, которые могут быть записаны в поле данного типа. Мы поговорим о них потом.

Посмотрим еще раз на рис. 1.2. Представленная там таблица имеет пять полей. А какого они типа? Давайте подумаем.

- Поле категории (`category`) получит строковый тип, поскольку название категории представляет собой слово. (Для него еще нужно указать предельную длину, но это можно сделать и потом.)
- Поле наименования товара (`name`) — также строкового типа. (Ему тоже следует задать предельную длину.)
- Полю описания товара (`description`) мы дадим тип memo. Такого рода данные могут иметь достаточно большой размер.
- В случае поля цены (`price`) вариант один — число с плавающей точкой. (Впрочем, раз цены на наши товары указываются в целых рублях, то можно использовать и целочисленный тип.)
- В случае поля признака, указывающего на наличие товара (`in_stock`), у нас тоже нет особого выбора — логический тип подходит для этого наилучшим образом.

Большинство форматов баз данных позволяют задать для поля *правила* — характеристики, которым должны удовлетворять записываемые туда данные. Такими условиями могут быть:

- обязательное наличие в поле какого-либо значения (*обязательное поле*);
- значение, которое должно быть помещено в поле при создании новой записи (*значение поля по умолчанию*);
- дополнительные условия (например, диапазон значений, в которые должно укладываться число).

Осталось только сказать, что набор полей с их именами, типами данных и условиями называется *структурой* таблицы. Сами реальные данные — содержимое полей и записей — в структуру не входят.

Индексы и ключи

Предположим, что мы создали таблицу, изображенную на рис. 1.2, заполнили ее данными и теперь пишем приложение, которое выводит на экран список товаров. И нам требуется отсортировать товары по какому-либо полю — например, по цене. В этом случае СУБД будет вынуждена:

1. Прочитать из базы данных все записи и поместить их в оперативную память, создав особый список.
2. Создать в памяти еще один список, содержащий все значения поля, по которому выполняется сортировка.
3. Переупорядочить эти значения, чтобы они шли по нарастанию или убыванию, и создать на их основе еще один список — третий по счету.

4. Соответственно переупорядочить записи таблицы и поместить их в отдельный список, который станет уже четвертым.

Если записей в таблице мало, этот процесс не займет много ни времени, ни оперативной памяти. А если записей там уже пара сотен?

Чтобы ускорить обработку записей, мы можем указать СУБД создать в базе отдельный массив данных, включающий все значения определенного поля таблицы, которые уже упорядочены нужным нам образом, и ссылки на соответствующие им записи. Понятно, что сортировка в этом случае будет выполняться много быстрее — ведь эти значения уже отсортированы, и СУБД остается лишь:

1. Прочитать из базы данных содержимое этого массива.
2. Прочитать из базы данных все записи таблицы.
3. Соответственно переупорядочить записи.

В этом случае, помимо ускорения обработки, потребуется еще и заметно меньше оперативной памяти — ведь будут созданы три списка, а не четыре, как ранее.

Такой список значений называется *индексом*, а поле, значения которого хранятся в индексе, — *индексированным*.

Индексы поддерживаются абсолютно всеми форматами баз данных и используются очень часто. В самом деле, индекс занимает немного места на диске и в памяти, а ускоряет операцию сортировки очень заметно. Единственный недостаток: при добавлении, изменении или удалении любой записи СУБД будет вынуждена соответственно изменить индекс, что отнимает некоторое время. Поэтому не стоит без необходимости создавать слишком много индексов.

Кроме сортировки, индексы также могут помочь при выполнении фильтрации записей. СУБД считывает индекс в память, ищет в нем значения, удовлетворяющие заданному критерию, и извлекает нужные записи из таблицы. Просто и быстро!

Изначально, при открытии таблицы, СУБД не считывает ни один индекс — они задействуются только при сортировке и фильтрации. Но имеется возможность сделать один из индексов загружаемым при открытии таблицы — при этом таблица будет изначально отсортирована согласно этому индексу. Такой индекс называется *ключевым*, или *ключом*, а задействованное в нем поле — *ключевым*. Ключевой индекс может быть только один на всю таблицу.

Ключевое поле должно удовлетворять следующим условиям:

- оно должно содержать значение (т. е. быть обязательным полем);
- оно должно содержать уникальные в пределах таблицы значения (быть уникальным полем).

Обычно в качестве ключевого применяется поле типа счетчика (см. табл. 1.1). Такие поля подходят для этого наилучшим образом. (Хотя, конечно, никто не мешает нам применить для этого поле любого другого типа.)

Ключевые индексы используются для того, чтобы однозначно идентифицировать какую-либо запись для изменения хранящихся в ней значений и ее удаления. Они также применяются для установления межтабличных связей.

Связи

Кстати, поговорить о межтабличных связях сейчас самое время. И вот почему...

Давайте посмотрим на таблицу, что показана на рис. 1.2. В частности, на поле `category`, где хранится категория товара. Чем примечательны хранящиеся в нем значения? И чем неоптимален такой способ указания категории?

Тем, что в этом поле записывается само ее название. Во-первых, оно довольно длинное и, соответственно, занимает немало места в базе данных. (Да, сейчас у нас категории имеют короткие названия. Но это сейчас...) Во-вторых, при вводе мы можем по ошибке указать название категории неправильно и тем самым нарушить работу приложения. В-третьих, если решим изменить название какой-либо категории, нам придется перебрать все относящиеся к ней товары и внести нужные правки в соответствующие им записи, что отнимет много времени.

Простое и красивое решение предлагает нам сам формат реляционных баз, который представляет данные как набор связанных таблиц.

Создадим в базе еще одну таблицу — для хранения списка категорий (рис. 1.3). Она будет содержать следующие поля:

- `id` — уникальный идентификатор записи, тип — счетчик, ключевое;
- `name` — название категории, тип — строковый.

id	name
1	Веники
2	Метлы

Рис. 1.3. Таблица — список категорий товаров

Теперь удалим из таблицы-списка товаров поле `category` и создадим в ней поле с тем же именем, но целочисленного типа. В этом поле будет храниться значение поля `id` таблицы-списка категорий, соответствующее данному товару. То есть вместо того, чтобы хранить в списке товаров сами названия категорий, мы будем помещать туда лишь ссылки на них.

Преимуществ у такого подхода два. Во-первых, мы храним в таблице не длинную строку, а короткое целое число, за счет чего размер базы данных станет меньше, а процесс ее обработки — быстрее. Во-вторых, мы не будем иметь никаких проблем, если вдруг захотим поменять название категорий, — ведь для этого нам потребуется исправить всего одну запись в таблице — списке категорий.

Можем себя поздравить — мы только что создали первую в нашей практике *связь* между таблицами (рис. 7.4).

В нашем случае одна запись списка категорий связана с произвольным количеством записей списка товаров. (В реальности так и бывает — в одну категорию могут входить множество товаров.) Это связь *один-ко-многим*. При этом таблица-список категорий будет *первичной*, или *родительской*, поскольку она, можно сказать, под-

чиняет себе связанные записи. А таблица-список товаров, напротив, станет *вторичной* или *дочерней*. Обе таблицы при этом станут *связанными*.

Поле вторичной таблицы, содержащее ссылки на записи первичной таблицы (у нас это поле `category`), называется *внешним индексом*. «Внешний» — потому что это поле внешнее по отношению к первичной таблице, а «индекс» — потому что практически всегда на основе этого поля создается индекс.

category	name	description	price	in_stock
1	В-1	Классическая конструкция из экологически чистых материалов	100	*
1	В-2М	Усовершенствованная конструкция, выполненная с применением особо прочной синтетики	200	
1	В-2000	Современная конструкция, выполненная с применением нанотехнологий	2000	*
2	М-1	Классическая конструкция из экологически чистых материалов	150	*
2	М-2014О	Выпущена в честь Зимней олимпиады в Сочи 2014 года	15000	*

id	name
1	Веники
2	Метлы

Рис. 1.4. Таблица-список категорий, связанная с таблицей-списком товаров

Правила построения реляционных баз данных требуют выделения одинаковых значений в отдельные связанные таблицы. В таком случае нам самим будет легче как поддерживать базу впоследствии, так и писать приложения, которые станут с ней работать.

Но что случится, если мы попытаемся удалить запись первичной таблицы, на которую ссылаются записи таблицы вторичной? СУБД просто не позволит это сделать, выведя нам сообщение об ошибке нарушения ссылочной целостности. Так что разорвать связь между записями, случайно или преднамеренно, у нас не получится.

Еще мы можем создать между таблицами связь типа *один-к-одному*, когда на одну запись первичной таблицы может ссылаться только одна запись таблицы вторичной. Если же мы попытаемся привязать к записи первичной таблицы еще одну запись вторичной, то, опять же, получим сообщение об ошибке. Однако такие связи на практике применяются довольно редко.

На этом ударный курс теории баз данных можно считать законченным.

Основные принципы разработки серверных Web-приложений

На очереди — не менее ударный курс теории программирования серверных Web-приложений.

Каждое более или менее сложное приложение (это относится к программам любого типа, не только к серверным) состоит из нескольких программных модулей, выполняющих различные задачи. И это понятно — ведь написать большое приложение, состоящее из одного модуля, очень трудно, если вообще возможно.

Все модули, из которых состоит серверное приложение, можно разделить на четыре разновидности: модели, контроллеры, шаблоны и служебные. Рассмотрим их поочередно.

Модели

Модель — это программный модуль, входящий в состав приложения, который служит своего рода посредником между остальными его модулями и базой данных. Или, говоря другими словами, модель — суть представление базы данных, ее таблиц, полей, индексов и связей в терминологии языка программирования, на котором пишется данное приложение.

Модель выполняет следующие задачи:

- описывает таблицы базы данных и их структуру в терминологии используемого языка программирования. Благодаря этому мы можем получать данные из базы, не прибегая к сторонним средствам;
- представляет считанные из базы данные в терминологии используемого языка программирования. Так что мы, считав с помощью модели какую-либо запись таблицы, сможем обработать ее средствами выбранного нами языка, опять же, не привлекая сторонние инструменты;
- реализует механизм выборки данных, их фильтрации и сортировки;
- реализует механизм добавления в таблицы новых записей, а также правки и удаления существующих;
- следит за корректностью данных, позволяя обрабатывать возникающие ошибки средствами выбранного языка программирования;
- возможно, расширяет набор средств, предоставляемых принятым форматом баз данных, добавляя к нему дополнительные инструменты, которые созданы разработчиком приложения или сторонними программистами.

Если уж совсем коротко, то модель — наш пропуск в базу данных.

Приложение может включать в свой состав произвольное количество моделей. Обычно каждая модель соответствует определенной таблице в базе данных.

Отметим сразу, что модели в приложении всегда играют подчиненную роль. Они вызываются другими модулями, относящимися к другой разновидности — кон-

троллерам, когда им требуется обратиться к базе данных, и вызываются явно, указанием в программном коде особой команды.

Контроллеры

Контроллер — это модуль приложения, выполняющий непосредственно обработку данных. Это главная действующая часть приложения, его сердце.

Обязанности у контроллера следующие:

- выборка данных из базы посредством явно вызываемых моделей;
- обработка полученных данных: фильтрация и сортировка;
- получение данных, отправленных пользователем;
- занесение полученных от пользователя данных в базу, опять же, посредством явно вызванных моделей, или иная их обработка;
- запуск формирования на основе результата обработки данных Web-страницы, которую увидит посетитель сайта.

Приложение может содержать произвольное количество контроллеров. Каждый контроллер соответствует определенному действию, выполняемому приложением, — так, выборка списка товаров выполняется одним контроллером, а добавление нового товара в список — другим.

Каждый контроллер ставится в соответствие определенному интернет-адресу приложения. Например, контроллер, выводящий список товаров, ставится в соответствие интернет-адресу **/goods/**, а контроллер, который добавляет товар в список, — интернет-адресу **/goods/add/**. Отслеживанием запрошенных посетителем интернет-адресов занимается особый служебный модуль, входящий в программное ядро приложения, он же выполняет запуск нужного контроллера при обращении к «его» адресу.

Контроллеры в процессе работы загружают и запускают остальные модули: модели и шаблоны. Так что их вполне можно назвать хозяевами положения... или приложения...

Шаблоны

Шаблон — это модуль приложения, единственное назначение которого — принять подготовленные контроллером данные и сформировать на их основе результирующую Web-страницу. Если совсем коротко, то шаблоны занимаются выводом данных.

Если модели и контроллеры представляют собой программы, написанные на языке программирования, то шаблон — это фактически обычная Web-страница, созданная на языке HTML. За единственным исключением — в ее код вставлены особые теги (*теги шаблона*), которые указывают, какие данные и в каком формате следует сюда поместить.

Приложение может включать в свой состав произвольное количество шаблонов. Правило здесь очень простое — каждой Web-странице, формируемой приложением, соответствует свой шаблон.

Как и модели, шаблоны явно вызываются контроллерами, когда им потребуется вывести обработанные данные.

Служебные модули

Что касается служебных модулей, входящих в состав приложения, то они включают в себя:

- модуль, отслеживающий запрошенные посетителем интернет-адреса и запускающий соответствующие им контроллеры (*диспетчер*);
- модуль, обрабатывающий теги шаблонов, т. е. подставляющий на их место отформатированные указанным образом данные (*шаблонизатор*);
- модуль настроек приложения;
- разнообразные модули, выполняющие типовые задачи Web-программирования: разграничение доступа, кэширование и пр.

Служебные модули составляют так называемое *программное ядро* приложения. Оно обеспечивает само функционирование приложения, не зависит от структуры создаваемого сайта и выполняемых им задач, вследствие чего может быть использовано в самых разных сайтах, обычно пишется один раз и модифицируется крайне редко, как правило, лишь с целью существенно расширить его функциональность.

Служебные модули могут как вызываться явно, когда в них возникнет нужда, так и постоянно функционировать «за кулисами». Одни служебные модули задействуются в любом случае (например, диспетчер), а другие могут включаться и отключаться в настройках приложения в зависимости от того, наличествует ли в них необходимость или нет.

Осталось лишь сказать, что принцип построения приложения, или, говоря другими словами, его *архитектура*, когда функциональность разделяется между моделями, контроллерами и шаблонами, носит название *модель-контроллер-шаблон*. (В зарубежной литературе применяется термин *model-view-controller*, *MVC*.)

Что дальше?

В этой главе мы выяснили, что такое Web-приложения и какие преимущества они несут по сравнению со статичными Web-страницами, что такое база данных и что она хранит. Еще мы узнали об архитектуре построения приложений модель-контроллер-шаблон и познакомились с моделями, контроллерами и шаблонами.

Следующая глава будет целиком посвящена языку программирования Python, на котором мы будем писать наши Web-приложения. Язык этот очень прост, в чем мы скоро и убедимся.

ГЛАВА 2

Язык программирования Python

В предыдущей главе мы рассмотрели основные принципы серверного Web-программирования и теорию баз данных. В этой главе мы приблизимся к практике, приступив к изучению языка программирования Python.

Python — высокоуровневый, объектно-ориентированный, тьюринг-полный язык программирования, одинаково хорошо подходящий для разработки как простейших командных скриптов, так и сложных настольных и Web-приложений. В комплекте с ним поставляется богатейшая стандартная библиотека, включающая мощные средства для обработки текстов, поддержки шифрования, работы с файлами, реализации обмена данных через Интернет и многое другое.

Но нас пока что интересуют базовые возможности Python, его синтаксис, поддерживаемые им типы данных, управляющие структуры и инструменты для работы с классами и объектами. Ими-то мы здесь и займемся.

Интерактивный интерпретатор Python

Разговор об этом языке будет сопровождаться большим количеством примеров. Чтобы проверить их в действии, мы можем использовать *интерактивный интерпретатор* Python, входящий в комплект его поставки.

После установки Python на компьютер в системном меню **Пуск** появится группа с именем вида **Python** ***<номер версии без последней цифры>***. (Например, у автора, установившего версию 3.3.4, эта папка носит имя **Python 3.3**.) В ней имеется ярлык **IDLE (Python GUI)**, который и запускает интерактивный интерпретатор.

Окно этой программы показано на рис. 2.1. Оно содержит большую область редактирования, куда вводится Python-код, и вследствие этого напоминает окно текстового редактора.

Введем в это окно следующее выражение:

```
2 + 3
```

и нажмем клавишу <Enter>. Тем самым мы дадим Python указание вывести на экран результат сложения чисел 2 и 3. Результат этот, равный 5, появится в следующей строке.

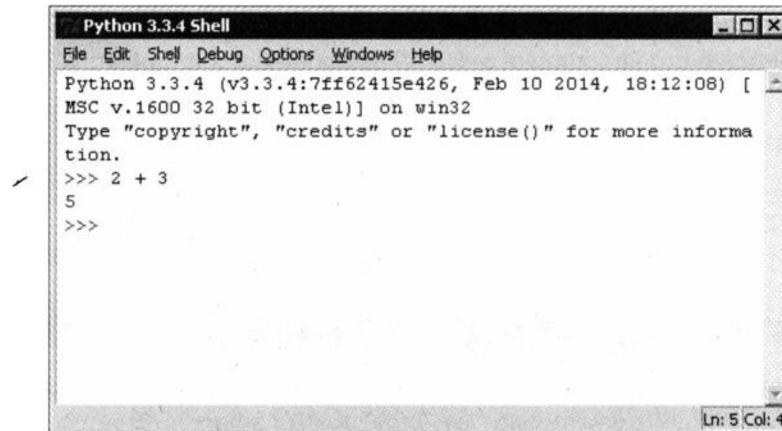

Рис. 2.1. Окно интерактивного интерпретатора Python

Что ж, по крайней мере, складывать числа этот язык умеет. Посмотрим, на что он еще способен...

Основные понятия Python

Начнем мы с самых простых понятий Python-программирования. Это выражение, оператор, функция и переменная.

Выражения

Ранее мы назвали команду `2 + 3`, введенную в окне интерактивного интерпретатора, *выражением*. И тем самым озвучили первый термин из всех, что нам предстоит запомнить.

Выражение в терминологии программирования — это команда, выполняющая законченное действие. Таким действием может быть вычисление некоего значения (как в нашем случае), создание какой-либо структуры данных, команда, управляющая выполнением программного кода, вызов функции или метода (о них мы поговорим потом) или что-то иное.

Любое выражение в Python должно завершаться символами возврата каретки и перевода строки, которые вставляются в программный код нажатием клавиши <Enter>.

Давайте рассмотрим примеры еще нескольких выражений.

- `3 * 4 + 8`

  Умножаем 3 на 4, прибавляем к получившейся сумме 8 и получаем 20. Операция умножения выполняется перед операцией сложения, т. к. она имеет больший приоритет.

- `5 / 6`

  Делим 5 на 6 и получаем длинный результат — 0.8333333333333334.

- `(3 + 7) * (6 + 4)`

  Складываем 3 и 7, складываем 6 и 4 и перемножаем получившиеся суммы. На выходе вполне ожидаемо окажется 100. Отметим, что операции сложения будут выполнены перед операциями умножения, т. к. они заключены в скобки.

- `round(1.8)`

  Округляем 1,8, что на выходе даст 2.

Операторы. Порядок выполнения и приоритет операторов

В самом первом нашем выражении мы использовали символ `+`, чтобы указать Python выполнить операцию сложения двух чисел, находящихся левее и правее этого символа. В данном случае `+` — это *оператор*, команда, выполняющая элементарное действие над одним или двумя значениями-*операндами* и выдающая, или, как говорят программисты, возвращающая, результат.

Также мы познакомились с операторами умножения `*` и деления `/`. А не знакомый нам пока оператор вычитания обозначается дефисом (`-`).

Как видим, операторы арифметических действий (*арифметические операторы*) обозначаются так же, как в обычных математических формулах. Это сделано для простоты.

Операторы в Python имеют *приоритет*, задающий порядок их выполнения. Так, операторы умножения и деления имеют более высокий приоритет, нежели операторы сложения и вычитания, поэтому выполняются в первую очередь (что мы и наблюдали ранее).

Операторы с одинаковым приоритетом выполняются в порядке слева направо.

Для изменения порядка выполнения операторов применяются круглые скобки — оператор, который требуется выполнить в первую очередь, помещается в них вместе с операндами:

```
(3 + 7) * (6 + 4)
```

Это выражение, как мы помним, даст в качестве результата 20.

Но если мы уберем скобки:

```
3 + 7 * 6 + 4
```

то Python сначала умножит 7 на 6, после чего прибавит к полученному произведению 3 и 4. На выходе мы получим число 49 — совсем другой результат!

Функции

Помимо операторов, в выражениях активно используются *функции*, которые выполняют уже более сложные действия над значениями, чем простое сложение или умножение. Одна из них нам уже знакома — это функция `round`, выполняющая округление числа с плавающей точкой: `round(1.8)`.

Обратим внимание на две особенности функций Python.

- Во-первых, значения, которые функция будет обрабатывать, или ее *параметры*, записываются за ее именем и заключаются в круглые скобки. Если таких параметров несколько, они разделяются запятыми.

  ```
  somefunction(1, 2, 345)
  ```

  Здесь мы передали гипотетической функции `somefunction` сразу три параметра.

  В некоторых функциях параметры имеют имена (*именованные параметры*). Чтобы передать таким функциям данные, используется вот такая запись:

  ```
  otherfunction(arg1 = 10, arg2 = 20)
  ```

  Здесь мы передали функции `otherfunction` два параметра с именами `arg1` и `arg2`.

- Во-вторых, если операторы всегда возвращают один результат, то функции могут вернуть сразу несколько. Во многих случаях это может оказаться полезным.

> **СОЗДАНИЕ СОБСТВЕННЫХ ФУНКЦИЙ**
>
> Функция `round` встроена в сам язык Python. Однако программист может создавать свои функции — как это делается, будет рассмотрено позже.

Переменные

Рассмотренные нами выражения вычисляли нужное нам значение сразу же, за один проход. Однако так получается далеко не всегда. Во многих случаях, чтобы вычислить какое-либо значение, приходится задействовать несколько идущих подряд выражений. Или, как вариант, некое значение, вычисленное в одном выражении, может не выводиться на экран, а использоваться в другом выражении для вычисления другого значения.

И возникает потребность как-то сохранить это значение для будущих вычислений.

Для такого случая предназначены *переменные*, которые можно рассматривать как ячейки памяти, имеющие уникальные имена. В переменную можно записать любое значение, а впоследствии извлечь его и использовать в дальнейших вычислениях.

Переменная может хранить только одно значение. Если поместить в переменную другое значение, предыдущее будет утеряно.

Давайте создадим, или, как говорят профессиональные программисты, *объявим*, нашу первую переменную:

```
a = 10
```

Здесь мы поместили в переменную с именем `a` число 10.

Знак равенства `=`, стоящий между именем переменной и заносимым в нее значением, — это *оператор присваивания*. Он присваивает переменной, чье имя стоит слева, значение, стоящее справа.

Отметим, что при объявлении переменной никакого результата на экран не выводится, поскольку оператор присваивания не возвращает результата.

Теперь мы можем использовать только что объявленную переменную:

```
a + 5
```

и получим 15.

Объявление переменной

Перед тем как использовать переменную, ее следует объявить. Если мы попытаемся обратиться к еще не объявленной переменной, то получим сообщение об ошибке.

Мы можем присвоить переменной результат выполнения выражения:

```
b = 2 + 3 / 4
```

или результат, возвращенный функцией:

```
c = somefunction(1, 2, 345)
```

Мы также можем объявить в одном выражении сразу несколько переменных:

```
a, b, c = 10, 2 + 3 / 4, somefunction(1, 2, 345)
```

Как видим, имена объявляемых переменных и присваиваемые им значения разделяются запятыми. Первая переменная получит первое значение, вторая переменная — второе значение и т. д.

Python также предоставляет нам возможность удалить не нужную более переменную. Это делает оператор `del`. Имена удаляемых переменных перечисляются за ним и разделяются запятыми.

```
del a, b
```

Здесь мы удаляем переменные `a` и `b`.

В программе может быть объявлено сколько угодно переменных. Следует помнить лишь две вещи: во-первых, имена переменных должны быть уникальными, а во-вторых, переменные расходуют оперативную память, поэтому лишних переменных объявлять не следует.

Типы данных и операции с ними

Следующий наш урок посвящен стандартным типам данных, поддерживаемым языком Python, и операциям, которые мы можем выполнять над ними. Таких типов четыре, причем один из них имеет три разновидности. Мы также рассмотрим особое значение `None`, которое может нам пригодиться в некоторых случаях.

Числа

Разумеется, любой язык программирования обязан поддерживать числа, или *числовой тип данных*. Python — не исключение.

Числа могут быть как целыми, так и с плавающей точкой. В последнем случае в качестве десятичного разделителя используется точка. Примеры задания чисел: `12`, `27`, `15000`, `2.3`, `3.3333333334`.

Если мы хотим записать целочисленное значение как число с плавающей точкой, то должны поставить в его конце десятичный разделитель (точку) и 0: `2.0`, `65.0`, `-3.0`. Если мы этого не сделаем, Python посчитает его целым числом.

Для записи чисел с плавающей точкой можно использовать экспоненциальную нотацию вида *<мантисса>e<порядок>*. Например: `1e3` (1000), `3e-4` (0,0003), `4.5e6` (4500000).

Для выполнения арифметических действий над числами мы можем пользоваться операторами, перечисленными в табл. 2.1. Многие из них нам уже знакомы.

***Таблица 2.1.** Арифметические операторы*

Оператор	Описание	Пример	Результат
`x + y`	Сложение	`10 + 5`	15
`x - y`	Вычитание	`3 - 8`	-5
`x * y`	Умножение	`2 * 4`	8
`x / y`	Обычное деление	`5 / 2`	2,5
`x // y`	Деление нацело	`5 // 2`	2
`x % y`	Остаток от деления	`5 % 2`	1
`-x`	Смена знака	`a = 3` `-a`	-3
`x ** y`	Возведение в степень	`5 ** 2`	25

Наконец, Python предоставляет пару полезных функций для обработки чисел. Они представлены в табл. 2.2.

***Таблица 2.2.** Функции для обработки чисел*

Функция	Описание	Пример	Результат
`abs(x)`	Получение абсолютного значения числа	`abs(-5)`	5
`round(x[, n])`	Округление числа. Количество знаков задается вторым параметром; если он не указан, число округляется до целого	`round(3.5)` `round(3.337, 2)`	4 3,34

Строки

В списке самых популярных типов данных строки оспаривают первое место с числами. Неудивительно, что Python предоставляет мощные средства для обработки *данных строкового типа*.

Запись строк

Строки заключаются либо в одинарные, либо в двойные кавычки.

```
'Python'
"Django 1.6.2"
```

Если строка заключена в одинарные кавычки, в ней не допускаются символы одинарных кавычек. А если строка заключена в двойные кавычки, в ней не допускаются символы двойных кавычек. Например, такие строки:

```
'Python '3.3.4''
"Django "1.6.2""
```

приведут к возникновению ошибки.

Если же нам все же потребуется вставить такой символ в строку, мы предварим его обратным слешем \:

```
'Python \'3.3.4\''
"Django \"1.6.2\""
```

Комбинация символов \n помещает в строковое значение символы возврата каретки и перевода строки:

```
'Python\nDjango'
```

В этом случае слово `Django` будет выведено в следующей строке.

Имеется также возможность задать строковое значение, которое представляет собой целый фрагмент текста, разбитый на строки. Такое значение предваряется комбинацией символов `'''` или `"""`, записывается, начиная со следующей строки, и завершается также комбинацией символов `'''` или `"""`, которая должна быть записана в следующей строке:

```
'''
Web-программирование:
  Python
  Django
'''
```

Это значение будет выведено на экран следующим образом:

```
Web-программирование:
  Python
  Django
```

Если записать подряд несколько строковых значений:

```
"Python" " 3.3.4"
```

они будут слиты в одну строку:

```
Python 3.3.4
```

Отметим, что таким образом можно объединить лишь сами строки, но никак не строковые значения, хранящиеся в переменных. Так, если мы запишем:

```
s1 = "Python"
s2 = " 3.3.4"
s1 s2
```

то получим сообщение об ошибке.

Кстати, процесс объединения нескольких строк в одну в программировании называется *конкатенацией*.

Обработка строк

Для работы со строками Python предоставляет довольно богатые стандартные средства (и еще более богатые инструменты, включенные в состав стандартной библиотеки).

Прежде всего, это оператор конкатенации `+`. Да, он записывается так же, как оператор арифметического сложения! И работает как с самими строками, так и со строковыми значениями, хранящимися в переменных:

```
"Python" + " 3.3.4"
s1 = "Python"
s2 = " 3.3.4"
s1 + s2
```

Строки можно умножать на целые числа. В этом случае результат будет представлять собой строку, повторенную соответствующее число раз:

```
s1 * 2
```

Результатом станет строка `PythonPython`.

Мы можем получить любой символ строки, просто указав его номер (*индекс*) в квадратных скобках после имени строковой переменной. Нужно только иметь в виду, что символы в строке нумеруются, начиная с нуля:

```
s1[0]
```

Это выражение выведет первый символ строки `s1` — букву `P`.

```
s1[2] + s1[5]
```

А это выражение выведет строку `tn`, составленную из третьего и шестого символов строки `s1`.

Если указать отрицательный индекс, символы будут отсчитываться не с начала строки, как обычно, а с ее конца. Запомним, что в этом случае символы начинают нумероваться с –1:

```
s1[-2]
```

Выполнив это выражение, мы получим второй с конца символ строки `s1` — букву `o`.

Мы можем выделить из строки фрагмент (*подстроку*), указав индексы ее начального и конечного символа в формате <*строка*>[<*начало*>:<*конец*>]. Здесь *начало* — это индекс начального символа выделяемой подстроки, а *конец* — индекс символа, ко-

торый следует за ее конечным символом. Другими словами, начальный символ включается в выделяемую подстроку, а конечный — нет.

Если не указан индекс начального символа, в подстроку включаются все символы с начала строки. Если же не указать конечный символ, в подстроку будут включены все оставшиеся символы строки:

```
s1[1:4]
```

Результатом будет подстрока, включающая второй, третий и четвертый символы строки s1, — yth.

```
s1[3:]
```

Здесь мы получим подстроку со всеми символами строки s1, начиная с четвертого, — hon.

```
s1[:3]
```

А здесь — подстроку со всеми символами строки s1, заканчивая третьим, — Pyt. (Не забываем, что конечный указанный символ не включается в подстроку.)

Теперь нужно уяснить один важный момент. Строки в языке Python неизменяемы. Это значит, что у нас не получится, скажем, заменить один из символов строки с помощью выражения вида:

```
s1[0] = "S"
```

Попытавшись его выполнить, мы получим ошибку.

Для работы со строками нам могут пригодиться четыре функции, представленные в табл. 2.3.

Таблица 2.3. Функции для обработки строк

Функция	Описание	Пример	Результат
len(s)	Получение длины строки	len("Python")	6
int(s)	Преобразование числа, представленного в виде строки, в целое число	int("2") int("2.33")	2 2
float(s)	Преобразование числа, представленного в виде строки, в число с плавающей точкой	float("2") float("2.33")	2,0 2,33
str(x)	Преобразование числа в строку	str(2) str(2.33)	"2" "2,33"

Списки

При разработке Web-сайтов с применением Python и Django мы часто будем манипулировать разнообразными списками. В самом деле, перечни категорий товаров, наборы записей в гостевой книге и статей в блоге — все это списки.

Python предлагает развитые инструменты для обработки *данных списочного типа*, или просто *списков*. Такие списки могут включать произвольное количество позиций (*элементов*) любого типа. (Хотя обычно список включает однотипные элементы.)

Обычные списки

Рассказ о списках мы начнем с *обычных списков* — они применяются в Python-программах чаще всего.

Создать список очень просто — достаточно заключить все входящие в него элементы в квадратные скобки, разделив их запятыми.

- `ln = [2, 54, 7.5, 890]`

  Мы только что создали список, включающий четыре числа.

- `ls = ["D", "j", "a", "n", "g", "o"]`

  А этот список включает все символы названия библиотеки, знакомство с которой мы начнем в следующей главе.

- `lc = ["c"]`

  Мы создали список из одного элемента.

- `le = []`

  А это пустой список, не содержащий ни одного элемента.

- `lq = [ln, ls, lc, le]`

  И, наконец, список, чьи элементы представляют собой созданные нами ранее списки, которые в данном случае будут называться *вложенными*.

Мы можем получить любой элемент списка тем же образом, что и любой символ строки:

```
ln[1]
```

Здесь мы получим число 54 — второй элемент списка `ln`.

Тем же способом мы можем получить список, включающий указанные нами элементы другого списка:

```
ls[1:3]
```

На выходе будет список ["j", "a"], включающий второй и третий элементы списка `ls`.

Списки Python, в отличие от строк, являются изменяемыми. Так что мы можем с легкостью изменить значение любого из его элементов:

```
ln[2] = 8.11
```

Меняем значение третьего элемента списка `ln`.

Также мы можем удалить любой элемент списка, воспользовавшись уже знакомым нам оператором `del`:

```
del ls[-1]
```

Удаляем последний элемент списка `ls`.

Мы можем складывать списки, пользуясь оператором +:

```
ls + [" ", "1", ".", "6", ".", "2"]
```

в результате чего получится список, содержащий все элементы из списков-операндов.

А уже знакомая нам по строкам функция `len` позволит узнать длину списка — количество его элементов.

Кортежи

Список с несколько неуклюжим названием *кортеж* (tuple) отличается от обычного списка тем, что является неизменяемым. Создав его, мы не сможем в будущем не изменить значения его элементов, ни удалить их.

Кортежи создаются так же, как обычные списки, за исключением того, что вместо квадратных скобок здесь ставятся круглые:

```
tn = (2, 54, 7.5, 890)
```

Создаем кортеж из четырех элементов.

Впрочем, скобки можно и не ставить:

```
tn = 2, 54, 7.5, 890
```

Кортеж из одного элемента создается следующим образом:

```
ts = (3,)
```

То есть после значения единственного его элемента ставится запятая — так мы сообщим Python, что хотим создать именно кортеж.

А пустой, не содержащий элементов кортеж создается с помощью пустых скобок:

```
te = ()
```

Правда, не совсем понятно, зачем он нужен...

Важным достоинством кортежа является то, что он может быть присвоен сразу нескольким переменным, которые получат значения, хранящиеся в его элементах.

```
n1, n2, n3, n4 = tn
```

В этом случае переменная `n1` получит значение первого элемента кортежа `tn`, переменная `n2` — значение его второго элемента и т. д.

Операция присваивания кортежа нескольким переменным называется *распаковкой*.

В разделе, посвященном переменным, мы говорили, что можем в одном выражении объявить сразу несколько переменных, записав через запятую имена переменных и присваиваемые им значения. Фактически при этом мы создавали кортеж, который сразу же распаковывали.

Словари

Словарь — это особый список, элементы которого имеют заданные нами индексы. В качестве их индексов могут выступать числа, строки и кортежи, включающие

числа и строки; однако на практике для этого используются строки. Кстати, такие заданные программистом индексы называются *ключами* (не путать с ключевыми индексами, описанными в *главе 1*).

Словарь создается почти так же, как обычный список, за двумя исключениями. Во-первых, список элементов берется не в квадратные, а в фигурные скобки. Во-вторых, сами элементы записываются в формате *<индекс>*: *<значение>*:

```
d1 = {"name":"Python", "version":"3.3.4", "category":1}
d2 = {"name":"Django", "version":"1.6.2", "category":2}
```

Мы создали два словаря, каждый из которых имеет по три элемента со строковыми ключами "name", "version" и "category".

Пустой словарь создается указанием «пустых» фигурных скобок:

```
de = {}
```

Мы можем получить доступ к любому элементу словаря по его ключу:

```
v = d2["version"]
```

и изменить его значение:

```
d2["version"] = "1.6.3"
```

что говорит о том, что словари, как и обычные списки, являются изменяемыми.

Обычно словари применяются для хранения данных, относящихся к какой-то одной сущности. В нашем примере мы так и поступили, описав в каждом словаре сведения об используемых нами платформах разработки Web-приложений.

Присваивание списков. Ссылки

А теперь рассмотрим один крайне важный вопрос. Он касается присваивания списков.

Давайте создадим вот такой простой список:

```
l1 = [1, 2, 3]
```

и присвоим его другой переменной:

```
l2 = l1
```

Теперь сменим значение первого элемента списка на другое:

```
l1[0] = 6
```

И проверим, что хранится во втором списке, набрав в интерактивном интерпретаторе имя хранящей его переменной l2.

Что же мы увидим? А то, что значение первого элемента второго списка также изменилось на 6! Выходит, что в переменных l1 и l2 хранится один и тот же список!

Так и есть! Как только мы создаем список и присваиваем его переменной, последняя получает в качестве значения не сам этот список, а *ссылку* на него, которую можно считать указателем на фрагмент оперативной памяти, где он содержится.

А когда мы присваиваем значение переменной, где хранится список, другой переменной, туда копируется эта ссылка. В результате обе переменные фактически хранят один и тот же список.

Кстати, то же самое касается чисел, строк и логических величин (о них — чуть позже). При их присваивании другой переменной в нее копируется не сама величина, а лишь ссылка на нее.

В языке Python все типы данных, даже самые простые, наподобие чисел и строк, являются ссылочными.

Логические величины

Логическая величина — это признак вида «да-нет», «включено-выключено» или «истина-ложь». Обычно *данные логического типа* используются в особых выражениях, управляющих порядком выполнения выражений (мы рассмотрим их позже).

Запись логических величин

Логическая величина записывается с помощью *ключевых слов* (особых слов языка Python, применяемых для специальных целей) `True` и `False`. Первое слово означает «истина», второе — «ложь».

```
l1 = True
l2 = False
```

Но, как правило, логические величины являются результатом вычисления особых выражений, которые называются *логическими*. В них применяются операторы двух типов, которые мы сейчас рассмотрим.

Операторы сравнения

Операторы сравнения, как следует из их названия, сравнивают между собой два операнда. Каждый тип данных поддерживает свой набор операторов сравнения.

Операторы сравнения чисел перечислены в табл. 2.4.

Таблица 2.4. *Операторы сравнения чисел*

Оператор	Описание	Пример	Результат
`x == y`	Равно	`2 == 2`	True
		`3.5 == 6.8`	False
`x != y`	Не равно	`2 != 2`	False
		`3.5 != 6.8`	True
`x < y`	Меньше	`1 < 2`	True
		`34 < 27`	False
`x > y`	Больше	`1 > 2`	False
		`34 > 27`	True

Таблица 2.4 (окончание)

Оператор	Описание	Пример	Результат
x <= y	Меньше или равно	1 <= 2	True
		34 <= 27	False
		155 <= 155	True
x >= y	Больше или равно	1 >= 2	False
		34 >= 27	True
		155 >= 155	True

Вот примеры:

```
n1 = 30
n2 = 40
l1 = n1 + 2 > n2
l2 = n1 != n2
```

После выполнения этого кода в переменной l1 окажется значение False, а в переменной l2 — значение True.

Отметим, что операторы сравнения имеют меньший приоритет, чем арифметические, поэтому выполняются после них.

Для сравнения строк служат операторы == и !=.

```
s1 = "Python"
s2 = "Django"
l = s1 != s2
```

После выполнения этого кода в переменной l окажется значение True.

Списки поддерживают два оператора сравнения, указанные в табл. 2.5.

Таблица 2.5. Операторы сравнения списков

Оператор	Описание	Пример	Результат
x in l	Входит ли значение x в состав списка l	2 in [2, 3, 4]	True
		1 in [2, 3, 4]	False
x not in l	Отсутствует ли значение x в списке l	2 not in [2, 3, 4]	False
		1 not in [2, 3, 4]	True

Помимо этого, списки поддерживают все операторы сравнения, перечисленные в табл. 2.4. В этом случае Python выполняет сравнение элементов списков последовательно. Если списки имеют разное количество элементов, то сравнению подвергаются лишь те элементы большего списка, что имеются в меньшем списке; остальные элементы большего списка в расчет не принимаются:

```
d1 = [2, 3, 4]
d2 = [1, 2]
l1 = d1 == d2
```

В переменной l1 окажется False, т. к. оба элемента списка d1 не равны первым двум элементам списка d2.

```
l2 = d1 < d2
```

А переменной l2 будет присвоено значение True, т. к. оба элемента списка d1 меньше первых двух элементов списка d2.

Кстати, если условное выражение возвращает True, программисты говорят, что оно выполняется, в противном случае — не выполняется.

Логические операторы

Часто бывает необходимо задействовать в логическом выражении не одно сравнение, а сразу несколько, причем по определенным правилам. Скажем, может возникнуть необходимость указать, что два условия должны выполниться обязательно или, наоборот, должно выполниться лишь одно из заданных двух условий.

Для таких случаев Python предусматривает так называемые *логические операторы*. Всего их три, и увидеть их можно в табл. 2.6.

Таблица 2.6. *Логические операторы*

Оператор	Описание	Пример	Результат
l1 and l2	Возвращает True, если значения l1 и l2 равны True, и False в противном случае	2 < 3 and 4 >= 1	True
		2 < 3 and 4 <= 1	False
l1 or l2	Возвращает True, если значение l1 или l2 равно True, и False в противном случае	2 > 3 or 4 >= 1	True
		2 > 3 or 4 <= 1	False
not l	Возвращает True, если значение l равно False, и наоборот	not 4 >= 1	False
		not 4 <= 1	True

Вот примеры:

```
n1 = 30
n2 = 40
l1 = n1 + 2 > 10 and n2 / 2 <= 80
```

Здесь переменная l1 получит значение True.

```
l2 = n1 - n2 > 0 or not n1 / n2 != 1
```

А здесь переменная l2 получит значение False.

Значение *None*

Еще Python поддерживает особое значение `None`. Оно обозначает отсутствие данных любого типа.

```
n = None
```

Мы будем пользоваться этим значением нечасто и лишь в специальных случаях.

Преобразования типов

Мы почти закончили со стандартными типами данных. Осталось лишь рассмотреть один важный вопрос.

Если мы сложим два целых числа:

```
2 + 2
```

то получим целое число. А если сложить целое число и число с плавающей точкой? А если нам захочется сложить строку и число? Что произойдет в этом случае?

В этом случае Python выполнит *преобразование типов данных*. Оно подчиняется законам, которые мы сейчас рассмотрим.

При выполнении арифметических операций с целыми числами и числами с плавающей точкой все целые числа преобразуются в числа с плавающей точкой:

```
1 + 3.5 + 2.5
```

Получится число с плавающей точкой `7,0`.

В условных выражениях (о них мы скоро поговорим) выполняется преобразование чисел, строк и списков в логические величины. В значение `False` преобразуются:

- математический ноль;
- пустая строка;
- пустой список любой разновидности;
- значение `None`.

Все прочие значения преобразуются в `True`:

```
s = "Django"
if 2 < 3 and s:
  n = 1
else:
  n = 2
```

В переменной `n` окажется значение 1.

> **ПРЕОБРАЗОВАНИЕ ЛОГИЧЕСКИХ ВЕЛИЧИН**
> Преобразование логических величин выполняется только в условных выражениях.

Во всех остальных случаях преобразование типов не выполняется. Так что мы не сможем просто так, например, объединить строку и число — нам придется явно преобразовать последнее в строковый тип:

```
"Python " + str(3)
```

Управление выполнением кода. Управляющие выражения

Очень часто в программировании бывает необходимо выполнить или не выполнить какие-либо выражения при выполнении или невыполнении определенного условия или выполнить некий код несколько раз подряд. Для таких случаев Python предусматривает несколько разновидностей *управляющих выражений*, о которых сейчас и пойдет речь.

Блоки

Но сначала нужно сказать пару слов о *блоках*. Такое название носит фрагмент кода, входящий в состав управляющих выражений (а также объявлений функций и классов, но об этом потом).

Выражения, входящие в блок, выделяются отступом слева. Этот отступ создается при помощи пробелов и может иметь любую длину — важно лишь помнить, что все выражения блока должны иметь одинаковый отступ. Однако на практике длина отступа чаще всего равна двум или четырем пробелам:

```
s1 = "Python"
s2 = "Django"
  n1 = 30
  n2 = 40
  l1 = n1 + 2 > 10 and n2 / 2 <= 80
l = s1 != s2
```

Здесь мы создали блок из трех выражений, выделив их отступом.

Условные выражения

Условные выражения позволяют выполнить или не выполнить блок при выполнении или невыполнении какого-либо условия. Это условие задается в виде логической величины.

Формат логического выражения таков:

```
if <условие 1>:
  <блок 1>
[elif <условие 2>:
  <блок 2>]
. . .
[else:
  <блок else>]
```

Сначала проверяется значение *условие 1*, расположенное после ключевого слова `if` (в секции `if`). Если оно возвращает `True` (выполняется), выполняется *блок 1*, после чего выполнение условного выражения прекращается, и Python начинает обработку кода, следующего за ним.

Если же значение *условие 1* равно `False` (не выполняется), проверяется *условие 2*, стоящее после ключевого слова `elif` (в секции `elif`). В случае равенства его `True` обрабатывается *блок 2*, и выполнение условного выражения прекращается. Отметим, что секция `elif` может отсутствовать.

Если же *условие 2* тоже оказалось равным `False`, обрабатываются остальные секции `elif`, если они есть. Происходит это так же, как было описано ранее.

Может случиться так, что ни одно условие из присутствующих в секциях `if` и `elif` не будет выполнено. Тогда выполняется *блок else*, присутствующий в секции `else`, если, конечно, он есть. После этого обработка условного выражения заканчивается.

Лучше один раз увидеть, чем семь раз услышать. Поэтому рассмотрим несколько примеров условных выражений, чтобы понять их работу.

- Пример 1:

```
if s == "Python":
  c = 1
elif s == "Django":
  c = 2
elif s == "Apache":
  c = 3
else:
  c = 0
```

  Это выражение будет присваивать переменной `c` числовое значение в зависимости от значения переменной `s`. Так, если `s` содержит строку `"Python"`, `c` получит значение 1, а если `"Django"` — то значение 2. Если же `s` содержит значение, не совпадающее ни с одним из сравниваемых, `c` присваивается 0.

- Пример 2:

```
if s == "Python":
  c = 1
else:
  c = 0
```

  Здесь мы присваиваем переменной `c` значение 1, только если `s` содержит строку `"Python"`, во всех остальных случаях `c` получит значение 0.

- Пример 3:

```
if s == "Python":
  c = 1
```

  Самый «куцый» вариант условного выражения, содержащего лишь секцию `if`. Здесь, если `s` содержит строку `"Python"`, `c` присваивается 1, в остальных случаях значение этой переменной не изменяется.

Циклы

Циклы позволяют выполнить блок кода несколько раз — либо пока не перестанет выполняться определенное условие, либо пока в списке не кончатся элементы.

Цикл с условием

Цикл с условием будет раз за разом выполнять блок, пока заданное нами условие возвращает `True`. Как только оно вернет `False`, начинает выполняться код, следующий за циклом.

Формат цикла с условием очень прост:

```
while <условие>:
  <блок>
```

Поскольку здесь применяется ключевое слово `while`, циклы такого рода получили название циклов `while`. А выполняемый блок часто называют *телом цикла*.

```
l = []
i = 2
while i <= 1024:
  l = l + [i]
  i = i * 2
```

Этот цикл создает список, чьими элементами станут степени числа 2: [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024].

Цикл по списку

Цикл по списку выполняет блок столько раз, сколько элементов включает в свой состав указанный в нем список. Формат его записи таков:

```
for <переменная элемента списка> in <список>:
  <блок>
```

В *переменную элемента списка* каждый раз будет помещаться значение очередного элемента *списка*. Эта переменная будет доступна только внутри тела цикла.

Цикл по списку имеет неофициальное название цикла `for-in`, т. к. он создается с применением ключевых слов `for` и `in`.

```
ls = [2, 8, 32, 128]
ld = []
for el in ls:
  ld = ld + [2 ** el]
```

Этот цикл создает список `ld`, элементы которого будут представлять собой число 2, возведенное в степени, чьи значения будут извлекаться из списка `ls`: [4, 256, 4294967296, 340282366920938463463374607431768211456].

Дополнительные возможности циклов

Иногда бывает необходимо прервать исполнение цикла досрочно. Для этого применяется *оператор прерывания цикла* `break`. Встретив его, Python тотчас перестает выполнять цикл и начинает обрабатывать код, который следует за ним.

```
ls = [2, 8, 32, 128]
ld = []
```

```
for el in ls:
  if el > 100:
    break
  ld = ld + [2 ** el]
```

В этом случае список `ld` будет содержать только элементы 4, 256 и 4294967296.

Как видим, оператор `break` не принимает операндов и сам по себе представляет собой полноценное выражение.

Похожий на него *оператор перезапуска цикла* `continue` указывает Python пропустить все выражения тела цикла, что следуют за ним, и начать следующий проход цикла.

```
ls = [2, 8, 32, 128]
ld = []
for el in ls:
  if el < 100:
    continue
  ld = ld + [2 ** el]
```

А в этом случае список `ld` будет содержать лишь одно значение — 340282366920938463463374607431768211456.

Функции

Собственно, с функциями мы уже знакомы. Мы вкратце рассмотрели их, когда разбирали основные принципы Python-программирования. Тогда мы использовали функции, встроенные в сам этот язык, и обмолвились, что программист сам может объявить свои собственные функции. Настала пора узнать, как это делается.

Объявление функции

Для описания функции применяется ключевое слово `def`. А формат самого объявления такой:

```
def <имя функции>(<список параметров функции>):
  <блок - тело функции>
```

Имя функции должно быть уникальным. Параметры в списке разделяются запятыми, если же функция не принимает параметров, после ее имени ставятся пустые круглые скобки.

Чтобы вернуть из функции вычисленный в ней результат, используется *оператор возврата значения* `return`. Возвращаемое в качестве результата значение ставится после этого оператора:

```
def pov2(n):
  return 2 ** n
```

Мы только что объявили нашу первую функцию. Она принимает один параметр, возводит число 2 в степень, равную значению этого параметра, и возвращает полученный результат.

Вызовем ее:

```
a = pov2(3)
```

и получим в переменной a значение 8.

> **Объявление функции**
>
> Функция должна быть объявлена перед первым ее вызовом. Если мы попытаемся обратиться к еще не объявленной функции, получим сообщение об ошибке.

```
def somefunc(n1, n2):
  return n1 * 2, n2 / 2
```

А эта функция принимает сразу два параметра и возвращает в качестве результата кортеж из двух значений. Мы можем впоследствии распаковать его, присвоив двум переменным:

```
a, b = somefunc(4, 9)
```

Локальные переменные

Как и в обычном программном коде, мы можем объявлять в теле функции переменные. Как и в обычном программном коде, эти переменные могут содержать значения любого типа, и число их не ограничено.

Однако переменные, объявленные в теле функции (их, кстати, называют *локальными*), имеют некоторые отличия:

- локальные переменные существуют и доступны только внутри тела функции. Программный код, находящийся вне функции, получить к ним доступа не может;
- мы можем в теле функции получить доступ к переменным, объявленным «снаружи» (*глобальным переменным*), но только в том случае, если ранее не объявили локальную переменную с таким же именем, как у глобальной;
- если мы объявим локальную переменную, чье имя совпадает с именем глобальной переменной, то локальная переменная подменит, или *скроет*, собой глобальную.

```
CONSTANT = 2
def somefunc(n1, n2):
  a = n1 * CONSTANT
  b = n2 / CONSTANT
  return a, b
```

Здесь мы объявили глобальную переменную CONSTANT и функцию somefunc с двумя локальными переменными — a и b. Отметим, что в теле функции мы также можем получить доступ к глобальной переменной...

```
def otherfunc(n1, n2):
  CONSTANT = 3
  a = n1 * CONSTANT
```

```
  b = n2 / CONSTANT
  return a, b
```

...и даже записать в нее новое значение.

Значения параметров по умолчанию. Именованные параметры

Мы можем указать для какого-либо параметра функции значение по умолчанию. Это значение параметр получит, если он не был указан в вызове данной функции.

Значение параметра по умолчанию указывается после его имени через знак =.

```
def somefunc(n1, n2 = 3):
  return n1 * 2, n2 / 2
```

Как видим, здесь применяется синтаксис, аналогичный объявлению переменной.

Теперь, если мы вызовем нашу функцию, не указав значение второго параметра:

```
somefunc(4)
```

он получит заданное нами значение по умолчанию.

> **РАЗМЕЩЕНИЕ ПАРАМЕТРОВ, ИМЕЮЩИХ ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ**
>
> Параметры, имеющие значения по умолчанию, должны в объявлении функции располагаться после параметров, таких значений не имеющих. Нарушение этого правила приведет к возникновению ошибки.
>
> Например, такое объявление функции:
>
> ```
> def somefunc(n2 = 3, n1):
>   return n1 * 2, n2 / 2
> ```
>
> будет ошибочным.

Параметры, для которых указаны значения по умолчанию, автоматически станут *именованными*. И мы сможем обращаться к ним по именам.

```
def otherfunc (n, arg1 = 2, arg2 = 89, arg3 = "!!!"):
  . . .
otherfunc(6, arg2 = 487)
```

Именованные параметры — исключительно удобная вещь, особенно в функциях с большим числом параметров. Нам больше не нужно запоминать порядок следования параметров — достаточно запомнить их имена и значения по умолчанию.

Функции с произвольным количеством параметров. Необязательные параметры

И, наконец, Python позволяет нам объявлять функции, принимающие произвольное количество параметров, — как обычных, так и именованных.

Указать, что функция может принимать произвольное число обычных, неименованных параметров, можно, задав в ее объявлении единственный параметр вида *<имя>:

```
def somefunc(*args):
  . . .
```

В теле функции будет создана локальная переменная с именем, которые мы указали. Значением этой переменной станет кортеж, включающий значения всех параметров, переданных данной функции:

```
def somefunc(*args):
  a = args[0]
  b = args[1]
  . . .
```

Теперь при вызове данной функции мы можем передать ей сколько угодно параметров:

```
somefunc(2, 7, 90, 45.7, "Python")
```

Мы можем объявить функцию, которая будет принимать некоторое количество обязательных параметров и произвольное количество *необязательных*. Например:

```
def somefunc(n1, n2, *args):
  . . .
```

Эта функция примет два обязательных параметра: первый и второй по счету, остальные параметры, которых может быть сколько угодно, станут необязательными.

Если нам нужно объявить функцию, принимающую произвольное количество необязательных именованных параметров, мы укажем в ее объявлении параметр вида `**`*<имя>*. Тогда в ее теле будет создана локальная переменная с указанным нами именем, значением которой станет словарь. Ключами элементов этого словаря окажутся имена параметров, переданных при вызове функции, а самими элементами — их значения:

```
def somefunc(**kwargs):
  keys = kwargs.keys()
  a = kwargs[keys[0]]
  b = kwargs[keys[1]]
  . . .
```

В теле этой функции мы получаем список всех ключей полученного словаря и, пользуясь им, извлекаем из словаря все элементы. (О методе `keys` класса `dict` и самом этом классе мы поговорим потом.)

- `somefunc(arg1 = 2, arg2 = 7, arg3 = "Python")`

  Вызываем только что объявленную функцию, передав ей три именованных параметра.

- `def otherfunc(n1, n2, **kwargs):`
  `. . .`

  Эта функция примет два обязательных неименованных параметра и произвольное количество именованных.

```
otherfunc(2.6, "3.3.4", arg1 = 79, arg2 = "Python")
```

Вызываем ее.

- ```
  def otherfunc(n, *args, **kwargs):
      . . .
  ```

  А эта функция примет один обязательный именованный параметр, произвольное количество необязательных неименованных и произвольное же количество необязательных именованных.

> **Расположение обозначений *<имя> и **<имя>**
>
> Обозначения `*<имя>` и `**<имя>`, указывающие списки необязательных параметров, должны стоять в объявлении функции в самом конце списка параметров, причем обозначение `**<имя>` должно находиться после `*<имя>`. Нарушение этого правила приведет к возникновению ошибки.

Как видим, Python позволяет нам писать очень сложные функции. Если мы решим заняться созданием дополнительных библиотек, нам это должно пригодиться.

Классы и объекты

Обычно программы хранят данные, относящиеся к какой-либо одной сущности, в разных переменных. Например, мы будем хранить в разных переменных наименование товара, его категорию, описание, цену, признак того, есть ли товар в наличии, и всевозможные дополнительные данные (имя файла с изображением товара, процент скидки на него и пр.).

Соответственно, для манипулирования данными, которые относятся к одной сущности, в программах применяются и разные функции. Так, у нас, скорее всего, будет функция, рассчитывающая новую цену товара с учетом скидки, функция, выдающая сокращенное описание товара, и др.

Вследствие этого возникает желание свести и данные, и обрабатывающие их функции в некую единую сущность. Но как это сделать? Конечно, данные мы можем свести воедино, оформив их в виде словаря (см. ранее), но как добавить в него функции?

Решение есть. Мы можем использовать объекты.

Основные понятия и приемы работы

Объект — это сложная сущность, хранящая в себе как данные, так и программный код, который манипулирует ими. Данные представляют собой значения, хранящиеся в *свойствах* — своего рода переменных, являющихся «собственностью» объекта. А обрабатывающий эти значения программный код организован в виде *методов*, которые можно считать принадлежащими объекту функциями.

Как обычные значения принадлежат к определенному типу, так и объект должен принадлежать определенному классу. *Класс* описывает тип объекта — набор его свойств и методов.
  ```

Например, в составе стандартной библиотеки Python поставляется класс `Fraction`. Он позволяет хранить десятичные дроби и производить над ними всевозможные действия: сокращение, сложение, вычитание и пр.

```
from fractions import Fraction
```

Здесь мы импортируем класс `Fraction` из модуля `fractions`. (О модулях Python и импорте из них типов мы поговорим в конце этой главы.)

Чтобы создать объект какого-либо класса, мы запишем имя этого класса, поставим после него круглые скобки, в которых поместим параметры создаваемого объекта. Получится нечто похожее на вызов функции. Результат этой функции — готовый объект — мы присвоим переменной или используем в дальнейших вычислениях.

Параметрами создаваемого объекта класса `Fraction` будут числитель и знаменатель дроби:

```
f = Fraction(8, 6)
```

В переменной `f` окажется объект класса `Fraction` — дробь 8/6.

Для доступа к свойствам и методам объекта мы запишем имя переменной, где он хранится, поставим точку и укажем имя нужного свойства или метода.

Например, класс `Fraction` поддерживает свойства `numerator` и `denominator`, хранящие, соответственно, числитель и знаменатель дроби. Чтобы получить их значения, мы используем такие выражения:

```
a = f.numerator
b = f.denominator
```

В переменной `a` окажется значение числителя дроби `f`, а в переменной `b` — значение ее знаменателя. Они будут равны 4 и 3 соответственно — как видим, сам класс `Fraction` уже сократил нашу дробь.

Еще данный класс поддерживает метод `limit_denominator`. Он принимает в качестве единственного параметра знаменатель и возвращает новый объект того же класса, хранящий ближайшую к изначальной дробь с указанным знаменателем:

```
f2 = f.limit_denominator(2)
```

В переменной `f2` окажется объект класса `Fraction` — дробь 3/2.

Свойства `numerator` и `denominator` хранят данные объекта, а метод `limit_denominator` ими манипулирует. Вполне ожидаемо, что они вызываются у объектов. И поэтому носят название *свойств* и *методов объекта*.

Но существует и другая разновидность методов, которые вызываются не у объекта, а непосредственно у класса, — *методы класса*. (Свойства класса Python не поддерживает.) Обычно они создают другие объекты этого класса на основе переданных им параметров.

Класс `Fraction` поддерживает метод класса `from_float`. Он принимает в качестве единственного параметра число с плавающей точкой и возвращает новый объект данного класса, хранящий созданную на основе переданного числа дробь:

```
f3 = Fraction.from_float(3.5)
```

В переменной `f3` окажется объект — дробь 7/2.

Осталось лишь сказать, что объекты — это ссылочные типы, т. е. при присваивании объекта другой переменной в последнюю помещается ссылка на него:

```
f4 = f3
```

В результате переменные f3 и f4 будут указывать на один и тот же объект.

Объявление классов

Как и в случае функций, разработчик может объявлять свои собственные классы. Для этого используется следующий формат:

```
class <имя класса>:
  <блок объявлений свойств и методов>
```

Имя класса должно быть уникальным. Свойства объявляются так же, как и переменные, а методы — так же, как функции.

> **ОБЪЯВЛЕНИЕ КЛАССА**
>
> Класс должен быть объявлен перед созданием первого его объекта. Если мы попытаемся создать объект еще не объявленного класса, получим сообщение об ошибке.

Отметим, что каждый метод должен принимать, по меньшей мере, один параметр, который в списке должен стоять первым. Обычно он имеет имя self. Этот параметр получит в качестве значения ссылку на сам этот объект — мы используем его, чтобы получить доступ к свойствам и методам данного объекта. Значение в данный параметр подставляется самим Python при вызове метода, и нам самим этого делать не нужно.

```
class Square:
  width = 0
  height = 0
  def area(self):
    return self.width * self.height
```

Здесь мы объявили класс Square, хранящий сведения о прямоугольнике. Он поддерживает свойства width и height, где хранятся ширина и длина прямоугольника, и метод area, возвращающий его площадь.

```
sq = Square()
sq.width = 100
sq.height = 40
ar = sq.area()
```

Создаем объект только что объявленного класса, задаем значения размеров прямоугольника и получаем его площадь.

Объявить метод класса можно, предварив объявление самого метода декоратором @staticmethod. (*Декораторами* в Python называются команды, выполняющие особые действия над функциями или методами.) Этот декоратор должен быть указан в отдельной строке, находящейся перед строкой с объявлением метода. А сам

метод не должен принимать в качестве параметра ссылку на сам объект (поскольку этого объекта нет):

```
class Square:
  . . .
  @staticmethod
  def get area(width, height):
    return width * height
```

Мы объявили в классе `Square` метод класса, вычисляющий площадь прямоугольника на основе переданных ему значений ширины и высоты.

Ранее, экспериментируя с классом `Fraction`, мы передавали параметры создаваемых дробей прямо при создании объектов. С нашим же классом в его изначальном виде такой номер не пройдет.

Чтобы получить возможность указывать значения свойств объекта прямо при его создании, мы должны объявить в классе особый метод — *конструктор*. Он должен иметь имя `__init__`:

```
class Square:
  . . .
  def __init__(self, width, height):
    self.width = width
    self.height = height
```

Наш конструктор принимает три параметра: обязательный для всех методов первый параметр, который получит в качестве значения ссылку на сам этот объект, ширину и высоту прямоугольника.

```
sq = Square(100, 40)
ar = sq.area()
```

Так гораздо удобнее — не нужно писать код для занесения значений в свойства.

Наследование классов

Одной из замечательных особенностей классов является *наследование*. Мы можем создать класс на основе другого класса, и второй класс получит в «наследство» все свойства и методы первого.

Класс, созданный на основе другого класса, получит название *потомка*, или *дочернего*. Соответственно, класс, на основе которого создается другой класс, называется *родителем*, или *родительским*.

Для создания класса-потомка применяется такой формат:

```
class <имя класса>(<имя класса-родителя>):
  <блок объявлений свойств и методов>
```

Если мы объявим в классе-потомке свойства и методы, чьи имена совпадают с именами таковых у класса-родителя, то свойства и методы потомка перекроют своих родительских «тезок». И если со свойствами такие штуки проделывать не рекомен-

дуется, то в случае методов эту особенность можно использовать для расширения унаследованной потомком функциональности.

Мы можем объявить в потомке метод с тем же именем, что у аналогичного метода родителя, но выполняющий дополнительные действия и, возможно, принимающий дополнительные параметры. В нужном месте этого метода мы вызовем одноименный метод родителя, использовав синтаксис *<имя класса-родителя>.<имя метода>(<список параметров>)*. Это избавит нас от необходимости копировать код «старого» метода в «новый»:

```
class Cube(Square):
  z = 0
  def __init__(self, width, height, z):
    Square.__init__(self, width, height)
    self.z = z
  def volume(self):
    return self.area(self) * self.z
```

Здесь мы объявили класс `Cube`, хранящий сведения о кубе, как потомок объявленного ранее класса `Square`. Мы добавили в него свойство `z`, хранящее высоту куба, и изменили конструктор, добавив в него выражение для сохранения переданного в качестве параметра значения высоты. Обратим внимание, как мы вызвали конструктор класса-родителя, который выполнит за нас всю работу по сохранению значений ширины и высоты. А еще мы добавили метод `volume`, который вернет значение объема куба.

```
c = Cube(100, 40, 5)
a = c.area()
v = c.volume()
```

В переменной `a` окажется число 4000, а в переменной `v` — число 20000.

Может случиться так, что вызываемый метод объявлен не непосредственно в родителе, а унаследован им от одного из его родителей, и мы не знаем, от кого именно. Тогда для вызова такого метода применяется синтаксис вида:

```
super(<имя текущего класса>, self).<имя метода>([<параметры>])
```

Отметим, что в *параметрах* в этом случае не указывается `self`.

Например, для нашего класса `Cube` вызов конструктора родителя будет выглядеть так:

```
super(Cube, self).__init__(width, height)
```

Осталось лишь сказать, что класс может иметь несколько родителей (так называемое *множественное наследование*). В этом случае их имена перечисляются через запятую:

```
class Child(Parent1, Parent2, Parent3):
  . . .
```

Стандартные типы Python как объекты

На закуску — самое интересное. Оказывается, стандартные типы данных Python: числа, строки, списки и логические величины — суть объекты соответствующих им классов!

Так, все целые числа являются объектами класса `int`, а все числа с плавающей точкой — объектами класса `float`. Класс `float` поддерживает полезный метод `is_integer`, который возвращает `True`, если данное число фактически является целым, и `False` в противном случае; параметров он не принимает.

```
n = 4.5
n.is_integer()
```

Эти два выражения дадут в качестве результата `False`, т. к. число 4,5 не целое.

```
2.0.is_integer()
```

А здесь мы вызвали данный метод не у переменной, а у самого числа. И вполне ожидаемо получили `True`, т. к. значение `2.0` хоть и обозначено как число с плавающей точкой, но на самом деле является целым.

Строки являются объектами класса `str`. Этот класс поддерживает огромное количество методов. Так, не принимающий параметров метод `capitalize` возвращает копию текущей строки с прописной первой буквой и строчными остальными.

Обычные списки являются объектами класса `list`. Он поддерживает, в частности, не принимающий параметров и не возвращающий результатов метод `clear`, который удаляет все элементы списка, аналогичный метод `sort`, сортирующий элементы в списке, и метод `count`, принимающий в качестве единственного параметра какое-либо значение и возвращающий количество элементов списка, чье значение равно указанному нами:

```
l = [1, 2, 3]
n = 4
if l.count(n) == 0:
  l = l + [n]
```

Кортежи являются объектами класса `tuple`, а словари — объектами класса `dict`. Последний класс поддерживает методы `keys` и `values` — они не принимают параметров и возвращают список всех ключей и значений словаря соответственно.

Логические величины являются объектами класса `bool`.

Обработка ошибок. Исключения

Как мы ни стараемся избежать возникновения в программах ошибок, они все равно будут возникать. Да, многие ошибки — например, неверно набранное ключевое слово, можно выявить сразу же. Однако другие, такие как деление на ноль, могут возникнуть в любой момент, и предотвратить их получается далеко не всегда.

Но ведь мы используем Python, один из популярнейших языков программирования! А он не стал бы таким популярным, не будь в нем мощных средств обработки ошибок.

Начать следует с того, что в случае возникновения ошибки Python создает особый объект, называемый *исключением*. По классу этого объекта мы можем определить, что случилось, и принять соответствующие меры.

Но как это сделать? С помощью особого выражения, называемого *блоком обработки исключений*. Формат его записи таков:

```
try:
  <блок try>
except <класс исключения 1>:
  <блок except 1>
[except <класс исключения 2>:
  <блок except 2>]
. . .
[else:
  <блок else>]
```

Поскольку для создания такого выражения применяются ключевые слова `try` и `except`, его еще называют блоком `try-except`.

Код, в котором может возникнуть ошибка, оформляется в виде *блока try*. Код, который должен сработать в случае возникновения исключения определенного класса, помещается в *блоке except*, где после ключевого слова `except` указывается имя нужного *класса исключения*. Таких блоков может быть сколько угодно. А код, который должен выполниться в случае отсутствия ошибок, оформляется как *блок else*.

Как только в *блоке try* возникает ошибка, Python, как мы уже говорили ранее, генерирует исключение. И сразу же начинает просматривать имена *классов исключений*, указанных после ключевых слов `except`. Найдя нужный класс, он выполняет указанный за ним *блок except*, после чего начинает обрабатывать код, следующий за блоком обработки исключений.

Если же в *блоке try* не возникнет ошибок, будет выполнен *блок else*, и, опять же, начнет выполняться следующий за блоком обработки исключений код.

Список некоторых классов исключений, с которыми мы столкнемся в своей программистской практике, приведены в табл. 2.7.

Таблица 2.7. *Классы часто наиболее встречающихся исключений*

Класс исключения	Описание
`AttributeError`	Ошибка обращения к свойству или методу класса
`IndentationError`	Некорректный отступ
`IndexError`	Список не имеет элемента с таким индексом
`KeyError`	Словарь не имеет элемента с таким ключом
`NameError`	Переменная, функция или класс с таким именем не существует

Таблица 2.7 *(окончание)*

Класс исключения	Описание
NotImplementedError	Данный метод в классе не реализован
OverflowError	Слишком большое число
RuntimeError	Нераспознанная ошибка времени выполнения
SyntaxError	Ошибка в записи кода (*синтаксическая ошибка*)
TabError	Отступ задан табуляциями, а не пробелами
TypeError	Неверный тип параметра функции или метода; свойство или метод не поддерживается классом
UnicodeError	Возникла проблема при раскодировании строки, заданной в кодировке UTF-8
ValueError	Неверное значение аргумента оператора, параметра функции или метода
ZeroDivisionError	Деление на ноль

Вот пример:

```
n = 0
try:
  m = 10 / n
except ZeroDivisionError:
  m = 0
```

Здесь в *блоке try* в любом случае возникнет ошибка деления на ноль и будет сгенерировано исключение класса ZeroDivisionError. После чего выполнится соответствующий ей *блок except*.

Иногда возникает необходимость в случае возникновения какого-либо исключения ничего не делать, тем самым *подавив* это исключение. Для таких случаев удобно применять так называемый *пустой оператор* pass, который ничего не делает:

```
. . .
except ZeroDivisionError:
  pass
```

В одном *блоке except* можно указывать несколько классов исключений. Они заключаются в круглые скобки и разделяются запятыми:

```
. . .
except (IndexError, KeyError):
  . . .
```

Если какое-то исключение не было обработано в блоке, оно поступит на обработку во внешний блок обработки исключения или, если такового нет, в саму исполняющую среду Python. В последнем случае будет выведено соответствующее сообщение.

Осталось выяснить, как самим генерировать исключения. Для этого мы применим *оператор генерирования исключения* `raise`, за которым указывается выражение создания объекта нужного исключения:

```
raise IndexError("В списке нет элемента с таким индексом!")
```

Комментарии

Очень редко программист может разобраться в написанном им же самим коде спустя достаточно продолжительное время. Поэтому хороший стиль программирования требует документирования кода, для чего применяются *комментарии*. Они не обрабатываются Python и предназначаются исключительно для программиста.

Комментарии в Python должны предваряться символом решетки `#`:

```
n = 1
# Присваиваем переменной n число 1
m = 4 # Присваиваем переменной m число 4
```

Комментарии часто применяются при отладке программ. Так, если мы подозреваем, что какой-то фрагмент кода вызывает ошибку, мы можем превратить его в комментарий (закомментировать) и посмотреть, как программа работает без него. Потом, разобравшись, в чем проблема, мы его раскомментируем.

Модули. Импорт. Библиотека

Да этого момента мы занимались исключительно тем, что набирали код в окне интерактивного интерпретатора Python (см. рис. 2.1) и смотрели, что получится в результате. Но в реальном программировании нам придется сохранять весь набранный программный код в особых файлах, которые и составят наше приложение, и часто обращаться к другим файлам, где хранится код стандартной библиотеки.

Поэтому сейчас самое время поговорить о модулях, пакетах, операциях импорта и упомянуть о стандартной библиотеке.

Модули и пакеты

Модулем называется текстовый файл, в котором хранится программный код, написанный на языке Python. Этот код может объявлять переменные, функции и классы, которые могут быть использованы как внутри этого модуля, так и в других модулях. Размер модуля и количество объявленных в нем переменных, функций и классов не ограничены.

Содержимое файла модуля должно быть сохранено в кодировке UTF-8. Создать такой файл можно в любом текстовом редакторе, поддерживающем данную кодировку, например, в хорошо знакомом нам Блокноте.

Файл модуля должен иметь расширение py. Его имя без расширения станет именем самого модуля. Так, мы можем сохранить код, объявляющий классы `Square` и `Cube`, в файле square.py. Тем самым мы создадим модуль `square`.

Обычно в модуль сводят код, выполняющий сходные задачи. Так, в один модуль можно свести функции, выполняющие вывод списка товаров и сведения об одном товаре, поскольку обе они занимаются выводом товаров.

Каждый модуль — это вещь в себе. Все объявленные в нем сущности: переменные, функции и классы — изначально недоступны для других модулей, составляющих приложение. Чтобы получить из одного модуля доступ, скажем, к функции, объявленной в другом модуле, нам придется выполнить операцию импорта (о которой речь пойдет позже).

Если модулей в приложении много, имеет смысл структурировать их с применением *пакетов*. Обычно пакет включает модули, выполняющие схожие функции или разработанные одним программистом.

Пакет Python представляет собой обычную папку операционной системы, в которой хранятся файлы модулей. Имя такой папки станет именем пакета. Помимо этого, чтобы указать Python, что эта папка является пакетом, обязательно следует поместить в нее пустой файл с именем __init__.py.

Имеется возможность создавать и вложенные папки, формируя тем самым вложенные пакеты. Количество пакетов, равно как и степень их вложенности друг в друга, не ограничены.

Например, мы можем поместить наш модуль `square` в пакет `figures`, который, в свою очередь, вложить в пакет `geometry`, создав следующую структуру папок:

```
GEOMETRY
  __init__.py
  FIGURES
    __init__.py
    square.py
```

Здесь прописными буквами набраны имена папок, а строчными — имя модуля. И не забываем о служебных файлах __init__.py — они говорят Python, что данная папка является пакетом.

При первом выполнении модуля Python *компилирует* его, преобразуя в компактное внутреннее представление и удаляя весь необрабатываемый код, в частности, комментарии. Если мы исправим модуль, он будет автоматически перекомпилирован. Откомпилированные файлы модулей имеют расширение вида cpython-<*номер версии Python*>.pyc (у автора книги такие файлы имеют расширение cpython-33.pyc) и хранятся в папке __pycache__, вложенной в папку модуля.

Импорт

Ранее говорилось, что любой модуль Python — это вещь в себе, и ни один другой модуль не сможет использовать объявленную в нем сущность, пока не выполнит операцию импорта.

Операция *импорта* явно указывает, что мы хотим использовать в данном модуле сущность, объявленную в другом модуле. Выполняется она с помощью *выражений*

импорта. Такое выражение содержит *оператор подключения модуля* `import`. Имена импортируемых модулей записываются после этого оператора и разделяются запятыми.

При обращении к сущности, объявленной в другом модуле, мы запишем имя этого модуля, поставим точку и уже после нее укажем само имя нужной сущности. То есть напишем ее *полное имя*.

Скажем, импортировать классы `Square` и `Cube` из модуля `square` в другой модуль мы сможем, написав выражение:

```
import square
```

А для обращения к классу `Square` мы укажем его полное имя:

```
s = square.Square(100, 40)
```

Если же нужный модуль находится в пакете, то полное имя сущности будет включать имена всех пакетов, в которые последовательно вложен модуль, имя этого модуля и имя самой сущности, разделенные точками.

Например, если наш модуль `square` последовательно вложен в пакеты `geometry` и `figures`, нам для использования класса `Square` в другом модуле потребуется написать такие выражения:

```
import geometry.figures.square
s = geometry.figures.square.Square(100, 40)
```

Чтобы каждый раз не писать полные имена сущностей (которые могут быть очень длинными), мы можем выполнить операцию *импорта с присоединением*, указав выражение формата:

```
from <имя модуля> import <имена присоединяемых сущностей>|*
```

Имена присоединяемых сущностей разделяются запятыми. Если нужно присоединить сразу все сущности, вместо их списка мы укажем символ звездочки `*`.

Так мы присоединим и используем класс `Square`:

```
from geometry.figures.square import Square
s = Square(100, 40)
```

А так — оба класса и вообще все, что объявлено в модуле `square`:

```
from geometry.figures.square import *
s = Square(100, 40)
c = Cube(200, 20, 8)
```

Стандартная библиотека. Сторонние библиотеки

Ранее мы несколько раз упоминали о стандартной библиотеке Python. Самое время сказать о ней несколько слов.

Стандартная библиотека состоит из множества модулей, объединенных в пакеты, и поставляется в составе Python. Она включает большое количество функций и

классов, выполняющих различные типовые задачи программирования. Там мы можем найти инструменты для сложной обработки строк, объявления новых типов данных, средства для обмена данными по сети, шифрования и дешифрования, работы с файлами, разработки многопоточных приложений и многое другое.

Давайте для примера возьмем модуль `datetime`. В нем объявлен класс `date`, позволяющий хранить и обрабатывать значения даты:

```
from datetime import date
```

Импортируем класс `date` из модуля `datetime`:

```
now = date.today()
```

Метод класса `today` возвращает объект класса `date`, хранящий текущее значение даты:

```
birthday = date(1970, 10, 27)
```

Создаем еще один объект класса `date`, хранящий дату 27 октября 1970 года:

```
delta = now - birthday
days = delta.days
```

Получаем количество дней, прошедших между этими датами. (Его хранит свойство `days` класса `date`.)

Также в стандартной библиотеке объявлен класс `Fraction`, который мы рассмотрели ранее.

Стандартная библиотека хранится в папке Lib папки, в которую установлен Python. Полные имена сущностей, объявленных в стандартной библиотеке, формируются относительно этой папки.

Стандартной библиотекой Python мы более подробно займемся потом, когда начнем разрабатывать Web-приложения. Сейчас же нам нужно знать лишь то, что она есть и всегда придет к нам на помощь.

Помимо стандартной библиотеки, поставляемой в составе Python, мы можем загрузить и установить любое количество *сторонних библиотек*. Они разрабатываются силами сторонних программистов (отчего и получили свое название), а существует их столько, что мы можем без труда найти ту, что нам нужна.

Все сторонние библиотеки устанавливаются в папку site-packages, что автоматически создается в упомянутой ранее папке Lib. Полные имена сущностей, объявленных в сторонних библиотеках, формируются относительно этой папки.

Кстати, Django, которой мы займемся уже в следующей главе, — это тоже сторонняя библиотека Python.

Текстовый редактор Notepad++

Ранее говорилось, что для написания Python-кода можно использовать любой текстовый редактор, даже стандартный Блокнот. На практике же Блокнот для этого не очень удобен — ведь нам самим придется ставить отступы, формирующие блоки.

К счастью, существуют специальные текстовые редакторы для программистов, которые, помимо автоматического выставления отступов, реализуют также синтаксическую подсветку кода, проверку на ошибки в синтаксисе и многие другие полезные вещи. Автор книги предпочитает пользоваться бесплатным редактором Notepad++ (рис. 2.2), который можно найти на сайте **http://notepad-plus-plus.org**.

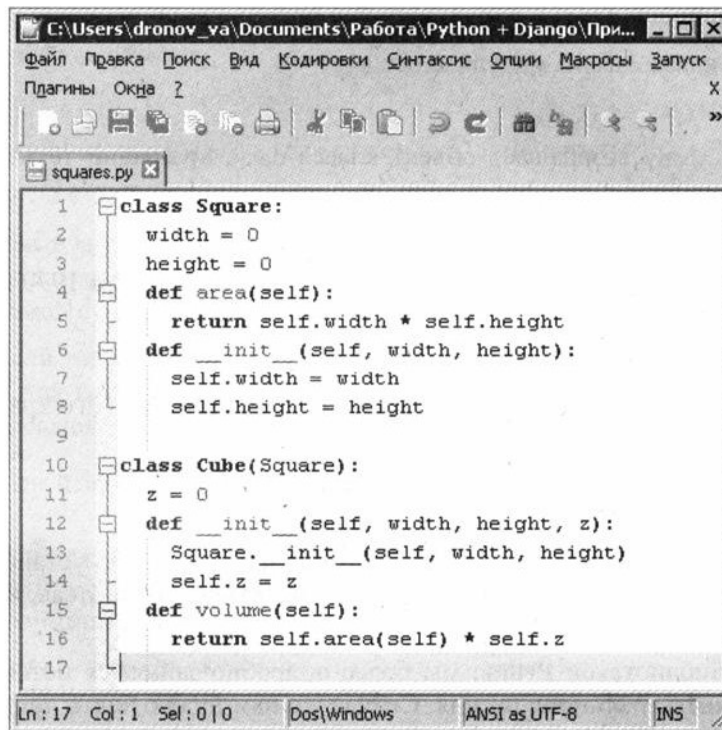

Рис. 2.2. Окно редактора Notepad++

По умолчанию этот редактор создает отступы с применением табуляции, в то время как Python требует лишь пробелы. Поэтому нам придется соответственно его настроить. Выберем в меню **Опции** (Settings) пункт **Настройки** (Preferences) и в левом списке открывшегося окна **Настройки** (Preferences) выберем пункт **Табуляция** (Tab Settings). Мы увидим то, что показано на рис. 2.3.

Проверим, выбран ли в списке **Настройка табуляции** (Tab Settings) пункт **[Default]**. Найдем флажок **Заменить пробелом** (Replace by space) и установим его. И не забудем нажать кнопку **Закрыть** (Close).

Чтобы сохранить набранный код в виде файла модуля Python, мы выберем в списке типов файлов диалогового окна сохранения файла пункт **Python file (*.py; *.pyw)**.

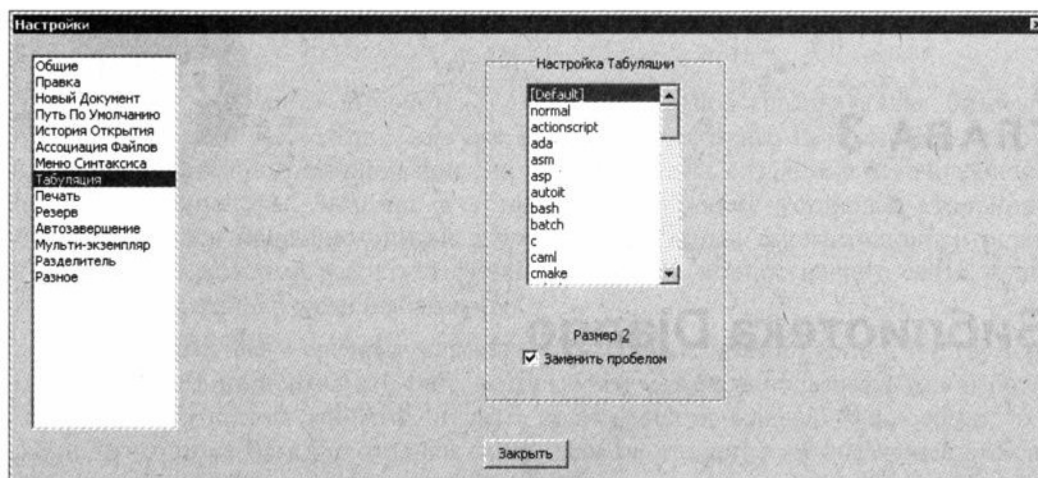

Рис. 2.3. Диалоговое окно **Настройки**; выбран пункт **Табуляция** левого списка

Что дальше?

Эта глава целиком посвящена языку программирования Python. Мы рассмотрели его основные понятия, типы данных и операции с ними, управляющие выражения, функции, классы и модули. Наконец, мы познакомились с текстовым редактором Notepad++, в котором и будем писать наш программный код.

Следующая глава посвящена библиотеке Django. Мы выясним, зачем она вообще нужна, что она дает нам как программистам, и выучим еще несколько терминов, которыми станем пользоваться в дальнейшем.

ГЛАВА 3

Библиотека Django

В предыдущей главе мы изучили язык программирования Python, на котором собираемся создавать свои Web-сайты. Как выяснилось, язык этот не так уж и сложен, но при этом предоставляет нам все нужные программисту инструменты.

Но для разработки сайтов нам понадобится кое-что еще. А именно, сторонняя библиотека Python, носящая звучное название Django. Вот о ней-то и пойдет сейчас разговор.

Библиотека Django — зачем она нужна?

Из *главы 1* мы знаем, что любой Web-сайт, построенный на основе Web-приложений, состоит из программных модулей трех разновидностей: моделей, контроллеров и шаблонов. Следовательно, нам достаточно написать все модели, контроллеры и шаблоны — и сайт готов. Так ли это?

Так, да не совсем.

Модели, контроллеры и шаблоны в любом случае не заработают сами по себе. Нам придется создать еще и необходимую программную инфраструктуру, которая выступит как костяк сайта, скрепит воедино все написанные нами модули, будет в нужный момент времени запускать их и поможет нам реализовать всяческие второстепенные задачи: разграничение доступа, кэширование и пр. Эта инфраструктура включает в себя:

- программное ядро, в частности, диспетчер (см. *главу 1*);
- шаблонизатор (автор специально выделил его из состава ядра, т. к. написать хороший шаблонизатор — крайне трудоемкая задача);
- базовую функциональность моделей и контроллеров;
- модули, реализующие второстепенные задачи Web-программирования — например, разграничение доступа.

Чтобы написать все это, как говорится, с нуля, потребуется много времени и сил. Времени и сил, которые мы можем употребить на что-либо более полезное, чем на

разработку того, что уже сделали за нас. Сделали и собрали в программный пакет с названием Django.

Django — это сторонняя библиотека для языка Python, реализующая базовую функциональность Web-сайта. Она уже включает в себя все необходимое, чтобы мы смогли начать программирование сайта, не занимаясь созданием перечисленной ранее инфраструктуры. Загрузив и установив ее (процесс установки сторонних библиотек Python подробно описан в *приложении 1*), мы сможем сконцентрироваться на коде, который реализует функциональность именно нашего сайта, — на его моделях, контроллерах и шаблонах.

Django содержит также средства для взаимодействия с базами данных, инструменты отладки, административный сайт, который мы можем использовать для работы с хранящимися в базе данными, и даже полнофункциональный Web-сервер, что крайне пригодится нам для отладки сайта. Нам не придется ни писать все это самим, ни прибегать к дополнительным программам.

Python и Django — вот «дуэт», что делает разработку даже сложных Web-сайтов такой простой!

Основные термины и принципы Django-программирования

Чтобы успешно использовать Django, нам нужно знать несколько новых принципов и терминов. Их немного, и они совсем несложны.

Проект

В терминологии Django *проектом* называется совокупность Web-приложений, составляющих один Web-сайт. Другими словами, проект — это сам Web-сайт, разработанный с применением Django.

Физически проект представляет собой обычную папку операционной системы. Имя этой папки станет именем проекта.

Структура папок и файлов проекта, формируемая при его создании самой Django, такова:

```
<ИМЯ ПРОЕКТА>
  manage.py
  <ИМЯ ПРОЕКТА>
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

Прописными буквами набраны имена папок, а строчными — имена файлов.

Как видим, во внешней папке, где хранятся файлы проекта (*папка проекта*), находится файл manage.py. Этот файл хранит код утилиты, которая позволит нам выпол-

нять различные действия над проектом, в частности, создавать в нем приложения — мы будем запускать этот файл из командной строки Windows.

В папке проекта присутствует еще одна папка с таким же именем. Это *пакет проекта* — он содержит модули, относящиеся к самому проекту. О том, что это именно пакет, говорит файл __init__.py, хранящийся в этой папке (о пакетах Python рассказывалось в *главе 2*).

В пакете проекта мы видим следующие модули:

- `settings` — хранит настройки проекта в виде набора переменных;
- `urls` — хранит сведения о привязке приложений к интернет-адресам;
- `wsgi` — служебный модуль, выступающий «посредником» между Web-сервером и проектом. Этот модуль потребуется нам в *главе 35*, когда мы будем заниматься публикацией нашего сайта.

В настройках проекта указываются сведения об используемой в приложениях проекта базе данных (или базах данных — мы можем указать их несколько), список активных приложений проекта, языковые параметры и некоторые другие данные. Отметим, что все эти настройки разделяются всеми приложениями, что входят в проект.

Папка проекта может находиться в любом месте файловой системы компьютера. Так что мы можем создать проект там, где нам удобно.

Приложение

Приложение Python входит в состав проекта и реализует функциональность одного из разделов сайта и всех его подразделов. Количество приложений в проекте не ограничено.

Физически приложение представляет собой пакет, папка которого находится в папке проекта (не в пакете проекта!). Имя этого пакета станет именем приложения, а сам пакет называется *пакетом приложения*.

Пакет приложения формируется самой Django при создании приложения. Изначально он содержит следующие модули:

- `models` — хранит код моделей, входящих в состав приложения;
- `views` — хранит код контроллеров, входящих в состав приложения;
- `admin` — хранит код, задающий параметры административного приложения, что входит в состав Django;
- `tests` — тестовый модуль.

Разумеется, мы можем, если возникнет такая необходимость, создать в пакете приложения другие модули, хранящие код моделей, контроллеров или дополнительных функций и классов, что мы задействуем в коде сайта. Django в этом плане никак нас не ограничивает.

Вот только все шаблоны, применяемые в сайте, нам в любом случае придется создать вручную. Django этого за нас не сделает.

Даже если приложение входит в состав проекта, это еще не говорит о том, что оно будет задействовано в сайте. Чтобы оно успешно работало, следует, во-первых, выполнить его привязку к интернет-адресу (об этом мы скоро поговорим), а во-вторых, указать его в списке активных приложений, что находится в модуле `settings` пакета проекта (см. ранее). Только после этого приложение станет *активным*.

Нужно сказать, что часть вспомогательных модулей Django реализована также в виде приложений (*встроенные приложения*). Таким встроенным приложением является, в частности, подсистема, реализующая разграничение доступа (разговор о ней пойдет в *главе 14*). Административный сайт (о нем мы поговорим в *главе 4*), с помощью которого мы можем работать с хранящимися в базе данными, также является приложением подобного рода.

Встроенные приложения либо также привязываются к интернет-адресу и, тем самым, формируют новый раздел сайта, либо работают постоянно, обеспечивая вспомогательную функциональность. В любом случае их также нужно указать в списке активных приложений, иначе они не будут задействованы.

Указание имен пакетов и модулей

Все полные имена пакетов и модулей, входящих в состав пакета проекта и отдельных приложений, указываются относительно папки проекта. Так, чтобы сослаться на модуль `views` пакета приложения `page`, следует указать полное имя `views.page`.

Привязка интернет-адресов

В *главе 1* мы узнали, что каждое приложение, входящее в состав сайта, запускается в ответ на обращение к определенному интернет-адресу, к которому оно было привязано. Единственное исключение здесь — встроенные приложения, с которыми мы только что познакомились и которые работают постоянно и не требуют такой привязки.

Каждому приложению ставится в соответствие своего рода виртуальная папка (об этом также говорилось в *главе 1*). Скажем, папку goods можно поставить в соответствие приложению списка товаров, выводящему список товаров, а папку guestbook — приложению гостевой книги, который выводит ее записи. Тогда посетитель, набрав интернет-адрес вида **http://www.somesite.ru/goods/**, попадет на список товаров, а, набрав **http://www.somesite.ru/guestbook/**, увидит гостевую книгу.

В библиотеке Django привязка интернет-адреса к приложению выполняется в модуле `urls` пакета проекта (см. раздел, посвященный проекту). Или, говоря другими словами, привязка адресов к приложениям выполняется на уровне проекта.

Но в реальности одно приложение может выполнять сразу несколько действий. Скажем, приложение списка товаров может выводить как и сам этот список, так и сведения о выбранном товаре, а приложение гостевой книги — как выводить гостевую книгу, так и добавлять в нее новую запись. Как это реализовать? Очень просто. Отдельным контроллерам приложений ставятся в соответствие виртуальные подпапки упомянутых ранее папок, формирующие подразделы данных разделов сайта.

В нашем случае мы привяжем подпапку / к контроллеру, выводящему список товаров, а подпапку good — к контроллеру, что выводит сведения об отдельном товаре.

Привязка подпапок, соответствующих подразделам сайта, к контроллерам приложения выполняется в модуле urls пакета приложения — уже на уровне приложения. (К сожалению, Django не создает этот модуль при формировании нового приложения, и нам придется сделать это самим.)

При привязке интернет-адреса к контроллеру мы можем указать, что последний должен принимать какие-либо данные. Эти данные будут переданы в составе интернет-адреса с применением метода GET. Например, поскольку мы собираемся реализовать вывод сведений о выбранном товаре, нам придется передавать соответствующему контроллеру идентификатор этого товара.

Структура Django-сайта

Как мы уже знаем, приложение Django реализует функциональность одного раздела сайта и всех его подразделов. Это достигается тем, что к интернет-адресу привязывается не само приложение, а его контроллер.

В связи с этим можно сформулировать правила структурирования сайтов, написанных на Django:

- один раздел сайта реализуется одним приложением. В число таких разделов входит и главная страница сайта;
- один подраздел раздела сайта реализуется одним контроллером, входящим в состав приложения;
- по возможности одно приложение не должно реализовывать функциональность, относящуюся к другому разделу сайта. Или, говоря другими словами, одно приложение не должно вторгаться в дела другого;
- однако административные задачи (например, наполнение сайта), по возможности, должны выполняться отдельным приложением, носящим название *административного*.

Соблюдать эти правила несложно, благо сам Python с его независимостью отдельных модулей и пакетов друг от друга подталкивает нас к этому. Вместе с тем, если нам понадобится в одном приложении использовать, скажем, модуль из другого приложения, мы всегда сможем это сделать, выполнив операцию импорта (см. *главу 2*).

Поддерживаемые форматы баз данных

Поскольку Django-сайт представляет собой набор приложений, все его данные хранятся в отдельной базе. Какой формат баз данных нам выбрать?

Django для хранения данных позволяет использовать базы разных форматов. Официально, самими разработчиками этой библиотеки заявлена поддержка PostgreSQL,

MySQL и Oracle, доступны также и сторонние библиотеки, разработанные третьими программистами и обеспечивающие поддержку других форматов баз данных.

Помимо этого, Django поддерживает формат баз данных SQLite. СУБД, работающая с такими базами данных, встроена в сам Python, так что никаких дополнительных программ и библиотек для реализации ее работы нам устанавливать не придется. А для нужд разработки сайтов ее возможностей вполне достаточно — если же потребуется, мы без труда сможем перенастроить сайт на работу с базой другого формата, того же MySQL.

Базы данных SQLite имеют и другое преимущество — нам не придется создавать их самим. Нужно будет лишь занести сведения о базе данных в настройки сайта, создать модели и выполнить простую команду синхронизации, после чего Django сама создаст базу, а в ней — все необходимые таблицы, индексы и связи.

Отладочный Web-сервер Django

В процессе разработки Web-сайта нам придется неоднократно запускать его, чтобы проверить, как он работает и работает ли вообще. Понятно, что для этого нужен Web-сервер, например, популярнейший Apache. Разработчики сайтов, применяющие другие языки и платформы (PHP, Zend, Yii и др.), просто вынуждены установить его на компьютер и соответственно настроить.

Но, поскольку мы используем язык Python и библиотеку Django, нам не нужен отдельный Web-сервер. В состав самой этой библиотеки входит *отладочный Web-сервер*, который устанавливается вместе с ней и не требует никакой настройки. Запустить его можно выполнением очень простой команды — мы без труда ее запомним.

Отладочный Web-сервер Django по умолчанию привязан к TCP-порту под номером 8000. Поэтому, чтобы запустить сайт на выполнение, мы должны набрать в Web-обозревателе интернет-адрес вида **http://localhost:8000/** или **http://localhost:8000/goods/**. (Впрочем, имеется возможность при запуске отладочного Web-сервера указать, чтобы он работал через другой порт.)

> ***Публикация готового сайта***
>
> Разработчики Django настоятельно не рекомендуют использовать отладочный Web-сервер для публикации готового сайта вследствие проблем с безопасностью. Поэтому, чтобы опубликовать Django-сайт, следует применить полнофункциональный Web-сервер, скажем, Apache.

Что дальше?

В этой главе мы рассмотрели теорию Django-программирования и построения Django-сайтов. Мы познакомились с проектами и приложениями, узнали о привязке интернет-адресов к приложениям, поддерживаемых форматах баз данных и отладочном Web-сервере, который пригодится нам при отладке. Можно начинать создание нашего первого Django-сайта!

ГЛАВА 4

Создание проекта и приложения Django

В предыдущей главе мы рассмотрели теоретические вопросы Django-программирования: что такое проект и приложение Django, как приложения соотносятся с интернет-адресами, какие форматы баз мы можем использовать для хранения данных сайта и что можем задействовать для его отладки. Поскольку нам не терпелось поскорее приняться за практику, мы особо на них не задержались (тем более что там и задерживаться-то было особо не на чем).

А сейчас настала пора практики. Мы узнаем, как создать и настроить проект Django, как указать сведения об используемой базе данных и как создать приложение. Мы также выясним, как запустить и остановить отладочный Web-сервер Django и как пользоваться встроенным в эту библиотеку административным Web-сайтом.

Создание проекта Django

Перед созданием проекта Django нам нужно выбрать место, где будет храниться его папка. Это может быть любая папка в файловой системе компьютера.

Запустим командную строку Windows. В ней, пользуясь соответствующими командами (они описаны в интерактивной справке к операционной системе), выполним переход в папку, где будет создан проект. И наберем команду следующего формата:

```
django-admin.py startproject <имя проекта>
```

не забыв нажать клавишу <Enter>. Через несколько секунд проект будет создан.

Файл django-admin.py хранит код административной утилиты Django, которая, в числе прочих действий, выполняет создание нового проекта. Командный ключ `startproject` выполняет создание нового проекта с именем, указанным за ним.

> ***Путь к файлу django-admin.py***
>
> Файл *django-admin.py* находится по пути *<папка, где установлен Python>\Lib\site-packages\django\bin*. Возможно, нам потребуется указать путь к нему, если мы не занесли его в список путей операционной системы (как это сделать, описано в *приложении 1*).

Хорошая новость: проект Django никоим образом не привязывается к конкретному пути операционной системы компьютера. Это значит, что мы не только можем создать его где угодно, но и имеем возможность для продолжения работы или публикации в Интернете переместить его в любую другую папку.

Запуск и останов отладочного Web-сервера

Создав проект, мы можем проверить, был ли он создан. Для этого мы запустим отладочный Web-сервер и попытаемся выйти на соответствующий проекту сайт.

Запускать отладочный Web-сервер удобнее из отдельного окна командной строки. Откроем его, выполним в нем переход в папку только что созданного проекта и наберем такую команду:

```
manage.py runserver
```

Как мы уже знаем из *главы 3*, файл manage.py хранит код служебной утилиты, предназначенной для работы непосредственно с проектом, которому она принадлежит. В частности, именно эта утилита запускает отладочный Web-сервер Django, для чего нам достаточно указать командный ключ `runserver`.

Как только мы нажмем клавишу <Enter>, в окне командной строки появится вот такой текст:

```
Validating models...

0 errors found
February 18, 2014 - 14:33:09
Django version 1.6.2, using settings 'sample.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Первая его строка говорит о том, что Django проверяет созданные в проекте модели приложений (которых у нас пока нет). По окончании проверки будет выведена пустая строка, а за ней — строка с количеством найденных ошибок (у нас — 0).

Если в результате проверки моделей не было найдено ошибок, в последующих строках появятся:

- сегодняшние дата и время;
- версия библиотеки Django и полное имя модуля настроек, относящегося к данному проекту (у автора это модуль `sample.settings`, т. к. проект носит имя `sample`);
- сообщение о запуске отладочного Web-сервера и его интернет-адрес (**http://127.0.0.1:8000**);
- сообщение о том, что для завершения отладочного Web-сервера следует нажать комбинацию клавиш <Ctrl>+<Break>.

Если мы теперь наберем в Web-обозревателе интернет-адрес отладочного Web-сервера, то увидим следующую страничку (рис. 4.1).

Рис. 4.1. Сообщение отладочного Web-сервера при пустом проекте

Если совсем коротко, она поздравляет нас с успешно установленной Django и приглашает создать первое приложение.

Завершим работу отладочного Web-сервера, переключившись на окно командной строки, где он был запущен, и нажав комбинацию клавиш <Ctrl>+<Break>.

Настройка проекта Django

Подождем пока создавать приложения в нашем новом проекте. Давайте сначала укажем для него необходимые настройки. В частности, нам потребуется задать сведения об используемой базе данных и параметры локализации (используемый на сайте язык и часовой пояс), а также выяснить на будущее, где хранится список активных приложений.

Откроем папку проекта. Найдем в ней папку, соответствующую пакету проекта, а в ней — модуль `settings`, где в виде набора переменных хранятся настройки сайта. И откроем файл этого модуля в текстовом редакторе Notepad++.

> ***Сохранение модуля settings***
>
> После внесения любых правок в настройки сайта следует сохранить модуль `settings`. Измененные настройки будут применены к сайту немедленно.

Сведения о базе данных

Как говорилось в *главе 3*, Django поддерживает хранение данных сайта в базах самых разных форматов. Мы для разработки сайта выбрали формат SQLite как самый простой в использовании и не требующий установки никаких дополнительных программ.

Параметры используемых баз данных хранит переменная `DATABASES`. Вот код, создающий значение этой переменной и сформированный Django при создании проекта:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Как видим, переменная DATABASES хранит словарь Python, каждый элемент которого также представляет собой аналогичный словарь.

Один элемент «внешнего» словаря задает сведения об одной базе данных. Ключ этого элемента фактически определит имя базы.

Если в нашем проекте задействована всего одна база данных, она должна называться default (как в представленном ранее коде). Если же наш проект оперирует несколькими базами, одна из них все равно должна иметь имя default — она станет базой по умолчанию, и к ней будет выполняться обращение, если имя базы не было указано явно.

Каждый элемент вложенного словаря определяет различные параметры, необходимые для подключения к базе данных. К таким параметрам относятся полное имя модуля, реализующего взаимодействие с базой, имя ее файла, адрес обрабатывающего эту базу сервера данных, имя и пароль, что будут использоваться для подключения к ней, и др.

Таких параметров очень много. Но в случае базы SQLite имеют смысл лишь два: полное имя модуля, реализующего взаимодействие с базой, и имя хранящего ее файла. Эти параметры задает сама Django при создании нового проекта.

Элемент внутреннего списка с ключом ENGINE задает полное имя модуля, выступающего как посредник между Django и базой данных. Для баз данных формата SQLite таким «посредником» выступает модуль django.db.backends.sqlite3.

Элемент внутреннего списка с ключом NAME задаст путь к файлу с базой данных. Мы можем указать его явно, в виде строки, или как результат, возвращенный функцией.

Кстати, по умолчанию он так и задан. Путь к файлу базы вычисляется функцией join из модуля os.path. Эта функция добавляет к пути, заданному первым параметром, имя файла, заданное вторым параметром, и возвращает полученный результат — полное имя файла. Оба параметра задаются в виде строк, и результат возвращается как строка. В нашем случае первым параметром этой функции является объявленная ранее в модуле settings переменная BASE_DIR, хранящая полученный в результате вычислений текущий путь к папке проекта, а вторым — имя файла db.sqlite3. Так что база данных сайта по умолчанию будет храниться в файле db.sqlite3, расположенном прямо в папке проекта.

Как правило, нет большого смысла указывать для файла базы другой путь — держа его в папке проекта, мы сохраним и данные, и код в одном месте, что удобно при разработке. А вот имя файла поменять смысл имеет, причем мы можем задать ему любое имя и даже расширение — скажем, назвать его site.dat или database.bin.

СУБД SQLite поддерживает и другие параметры, управляющие различными нюансами ее работы. Однако они очень специфичны, и их значения следует менять лишь в особых случаях.

Параметры локализации

Параметры локализации включают обозначение языка и временной зоны. Эти сведения используются, в частности, при сортировке записей в базах данных и выборе языка для вывода интерфейса встроенного административного Web-сайта Django (о нем будет рассказано в конце этой главы).

Параметры локализации задаются в переменных `LANGUAGE_CODE` и `TIME_ZONE` в виде строк.

```
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
```

Переменная `LANGUAGE_CODE` задает код языка сайта. В табл. 4.1 перечислены коды языков, с которыми мы будем иметь дело в большинстве случаев.

Таблица 4.1. Коды некоторых языков

Код	Язык
be-by	Белорусский (Беларусь)
en-gb	Английский (Великобритания)
en-us	Английский (США)
kk-kz	Казахский (Казахстан)
ru-ru	Русский (Россия)
uk-ua	Украинский (Украина)

Более полный список кодов языков можно найти по интернет-адресу **http://www.i18nguy.com/unicode/language-identifiers.html**.

Понятно, что, скорее всего, мы укажем для сайта русский язык, задав переменной `LANGUAGE_CODE` значение `ru-ru`.

Переменная `TIME_ZONE` служит для задания временной зоны. Различные ее значения для некоторых временных зон представлены в табл. 4.2.

Таблица 4.2. Обозначения некоторых временных зон

Временная зона	Город
Asia/Almaty	Алматы
Asia/Anadyr	Анадырь
Asia/Irkutsk	Иркутск
Asia/Kamchatka	Камчатка

Таблица 4.2 (окончание)

Временная зона	Город
Asia/Krasnoyarsk	Красноярск
Asia/Magadan	Магадан
Asia/Novokuznetsk	Новокузнецк
Asia/Novosibirsk	Новосибирск
Asia/Omsk	Омск
Asia/Sakhalin	Сахалин
Asia/Vladivostok	Владивосток
Asia/Yakutsk	Якутск
Asia/Yekaterinburg	Екатеринбург
Europe/Kaliningrad	Калининград
Europe/Kiev	Киев
Europe/Minsk	Минск
Europe/Moscow	Москва
Europe/Samara	Самара
Europe/Simferopol	Симферополь
Europe/Uzhgorod	Ужгород
Europe/Volgograd	Волгоград

Также можно задать значение UTC, обозначающее универсальное время. (Кстати, именно это значение задано в настройках сайта по умолчанию.)

Более полный список временных зон можно посмотреть на странице с интернет-адресом **http://en.wikipedia.org/wiki/List_of_tz_database_time_zones**.

Список активных приложений

Список активных приложений указывается в переменной INSTALLED_APPS в виде кортежа, состоящего из строк, значениями которых являются полные имена модулей этих приложений.

```
INSTALLED_APPS = (
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles',
)
```

Изначально здесь перечислены несколько встроенных приложений Django. Все они либо нужны для нормальной работы библиотеки, либо применяются в большинстве сайтов и поэтому включаются в список сразу при создании проекта.

- `django.contrib.admin` — реализует работу встроенного административного Web-сайта Django;
- `django.contrib.auth` — реализует работу подсистемы разграничения доступа (см. *главу 14*);
- `django.contrib.contenttypes` — обеспечивает автоматическую обработку моделей, например, при синхронизации (о синхронизации моделей с базой данных мы поговорим чуть позже);
- `django.contrib.sessions` — обеспечивает обработку сессий и необходимо для успешной работы подсистем разграничения доступа и хранения данных на уровне клиента (см. *главу 12*);
- `django.contrib.messages` — реализует работу подсистемы сообщений Django (см. *главу 12*);
- `django.contrib.staticfiles` — реализует функционирование подсистемы обработки статических файлов (см. *главу 8*).

Мы будем добавлять в этот кортеж новые приложения, как созданные нами, так и встроенными в Django.

Остальные настройки, присутствующие в модуле `settings`, пригодятся лишь в очень специфических случаях. Мы не будем их здесь рассматривать.

Синхронизация с базой данных

Мы уже знаем, что сразу при создании проекта некоторые встроенные приложения Django уже оказываются включенными в список активных приложений. Они нужны для нормальной работы сайта.

А для нормальной работы этих встроенных приложений им требуется база, в таблицах которой они будут хранить свои данные.

Проблема в том, что изначально у нас нет никакой базы данных, а даже если она и есть, то в ней отсутствуют нужные встроенным приложениям таблицы, индексы и связи. (Вспомним: мы ведь не создали базу, а лишь задали ее параметры в настройках проекта.) Поэтому нам прямо сейчас потребуется выполнить операцию синхронизации.

Операция *синхронизации с базой данных* осуществляет действия:

- создает базу данных (выполняется не для всех форматов баз);
- создает в базе необходимые структуры — таблицы, индексы и связи;
- возможно, заносит в таблицы некие изначальные данные.

> **НЕОБХОДИМЫЕ СТРУКТУРЫ ДАННЫХ**
>
> Создание необходимых структур данных выполняется всего один раз. Впоследствии, если таковые структуры уже присутствуют в базе, они создаваться не будут.

В качестве основы для создания самой базы и необходимых структур данных в ней используются, во-первых, заданные нами ранее настройки базы, а во-вторых, модели, являющиеся частью всех активных приложений проекта.

Выполнить синхронизацию очень просто. Откроем командную строку Windows, выполним переход в папку проекта и наберем такую команду:

```
manage.py syncdb
```

Командный ключ `syncdb` утилиты manage.py как раз выполняет синхронизацию.

Мы сразу же увидим многочисленные сообщения о создании в базе данных новых таблиц и индексов.

А теперь — внимание! Утилита manage.py выведет предупреждение о том, что в списке пользователей подсистемы разграничения доступа Django нет ни одного пользователя с административными полномочиями, и предложит нам создать такого пользователя. Настоятельно рекомендуется согласиться, введя в ответ на предупреждение строку `yes` и нажав клавишу <Enter>. (Если же мы не создадим администратора, то впоследствии не сможем войти на встроенный административный сайт Django.)

Далее нас попросят ввести имя пользователя-администратора, его адрес электронной почты и пароль, причем пароль нам потребуется ввести дважды — для надежности. Введем все это, не забывая нажимать клавишу <Enter>.

В конце концов мы увидим сообщение об успешном создании пользователя-администратора и еще несколько сообщений о ходе работы. И синхронизация завершится.

Создание приложения Django

Задав необходимые настройки только что созданного проекта и выполнив первую синхронизацию, мы можем создать наше первое приложение. Откроем командную строку Windows, перейдем в папку проекта и введем команду вида:

```
manage.py startapp <имя приложения>
```

Командный ключ `startapp` предписывает утилите manage.py создать в проекте приложение с именем, указанным после этого ключа.

Новое приложение создается быстро — не пройдет и секунды, как оно будет сформировано.

Теперь добавим его в список активных приложений. Откроем модуль `settings` из пакета проекта, найдем объявление переменной `INSTALLED_APPS` и изменим его код, добавив в кортеж пакет приложения.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'page'
)
```

Для примера автор создал приложение page, после чего добавил его в список активных.

Встроенный административный сайт Django

При разработке сайта нам часто понадобится заносить в базу какие-либо данные, необходимые для отладки. Разработчикам, использующим другие решения, потребуется привлекать для этого сторонние программы. Но нам этого делать не придется.

Библиотека Django включает в свой состав полнофункциональный *встроенный административный Web-сайт*, позволяющий просматривать содержимое созданных в составе приложений моделей, заносить в них новые данные и править уже существующие. Этот сайт активен по умолчанию, и чтобы запустить его, нам не надо будет выполнять никаких дополнительных действий.

Встроенный административный сайт привязан к виртуальной папке admin. Поэтому, чтобы войти в него, нам достаточно набрать в Web-обозревателе интернет-адрес **http://localhost:8000/admin/**.

> ***Локализация интерфейса административного сайта***
>
> Здесь описан русский интерфейс административного сайта. Чтобы его включить, следует указать в настройках проекта русский язык (как это сделать, было рассказано ранее).

Первое, что мы увидим, — страницу с формой входа (рис. 4.2). Введем в поле ввода **Имя пользователя** имя пользователя-администратора, а в поле ввода **Пароль** — его пароль и нажмем кнопку **Войти**. (Имя администратора и пароль мы указали при выполнении первой синхронизации с базой данных.)

При успешном входе мы попадем на страницу списка приложений (рис. 4.3). Она содержит набор таблиц, каждая из которых соответствует одному из активных приложений сайта: имя приложения указано в заголовке соответствующей таблицы, а строки таблицы соответствуют созданным в приложении моделям.

Так, на рис. 4.3 мы видим таблицу, соответствующую встроенному приложению Auth, которое обеспечивает работу подсистемы разграничения доступа. В этой таблице присутствуют строки **Группы** и **Пользователи**, представляющие две имеющиеся в данном приложении модели: первая модель хранит список групп пользователей, а вторая — список самих пользователей.

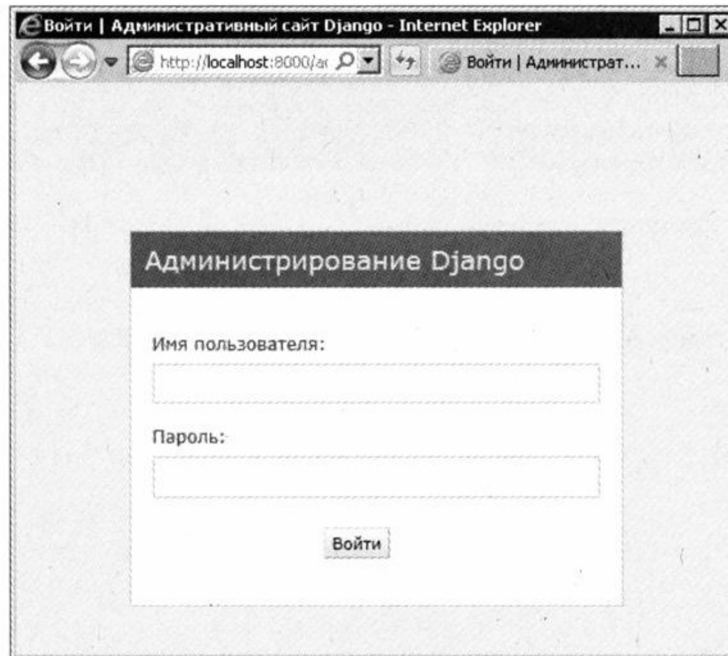

Рис. 4.2. Страница входа на административный сайт

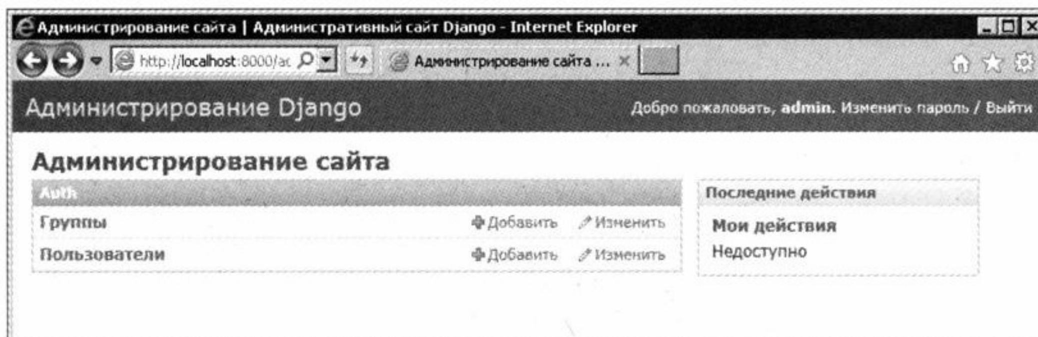

Рис. 4.3. Страница списка приложений

Чтобы просмотреть содержимое какой-либо модели, следует щелкнуть мышью на ее названии — это название представляет собой гиперссылку. Административный сайт в ответ выведет нам страницу содержимого модели (рис. 4.4).

Здесь мы видим таблицу, перечисляющую все записи выбранной нами модели. В нашем случае запись всего одна — она представляет пользователя-администратора, что мы создали при синхронизации (у автора этот пользователь носит имя `admin`).

Добавить в модель новую запись можно, нажав расположенную в верхнем правом углу кнопку **Добавить *<наименование сущности, хранящейся в модели>***. На экране появится страница добавления записи (рис. 4.5).

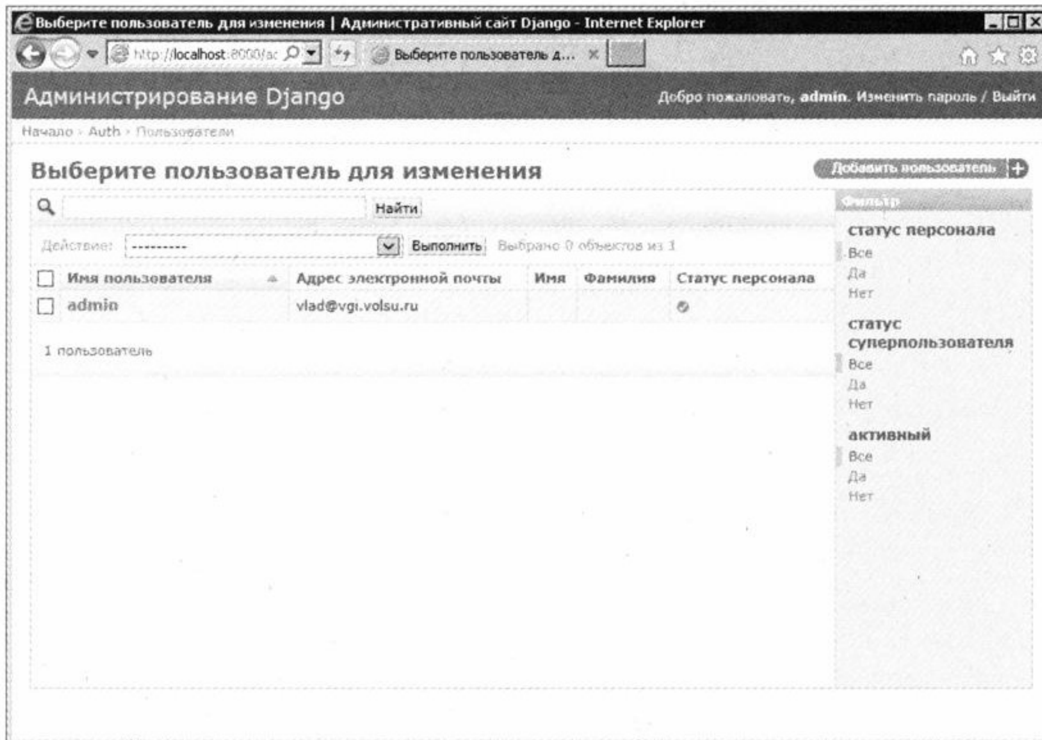

Рис. 4.4. Страница содержимого модели

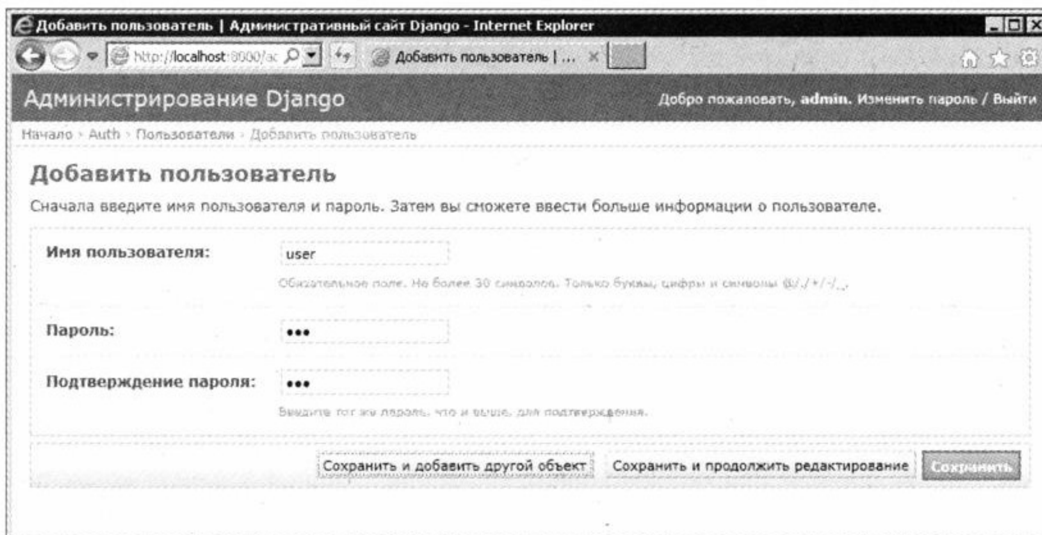

Рис. 4.5. Страница добавления записи в модель

Отметим сразу, что, если модель содержит большое количество полей, таких страниц добавления записи будет несколько, и эти страницы станут выводиться последовательно, одна за другой. К моделям такого рода относится и модель **Пользователи**, с которой мы сейчас работаем.

На страницах добавления записи все просто — вводим в элементы управления нужные значения и нажимаем одну из находящихся внизу кнопок:

- **Сохранить** — либо перейти на следующую страницу добавления записи, либо, если это последняя страница, сохранить новую запись и вернуться на страницу содержимого модели;
- **Сохранить и добавить новый объект** — либо перейти на следующую страницу добавления записи, либо, если это последняя страница, сохранить новую запись и подготовить сайт для добавления еще одной записи;
- **Сохранить и продолжить редактирование** — либо перейти на следующую страницу добавления записи, либо, если это последняя страница, сохранить новую запись и подготовить сайт для ее правки.

Исправить уже хранящуюся в модели запись тоже несложно. В таблице на странице содержимого модели (см. рис. 4.4) имеется столбец, содержимое ячеек которого представляет собой гиперссылки (в нашем случае — это строка в столбце **Имя пользователя**). Нам нужно всего лишь щелкнуть на той гиперссылке, что принадлежит нуждающейся в исправлении записи.

В результате мы перейдем на набор страниц изменения записи, похожих на уже знакомые нам страницы их добавления (см. рис. 4.5). В элементы управления этих страниц будут подставлены значения соответствующих полей выбранной нами записи. Внесем нужные правки и нажмем одну из перечисленных ранее кнопок.

Удалить ненужную запись несколько сложнее. На странице содержимого модели найдем запись, которую хотим удалить, выделим ее, установив расположенный в самом левом столбце флажок (кстати, так мы можем выделить для удаления сразу несколько записей), выберем в списке **Действие** пункт **Удалить выбранные** ***<имя модели>*** и нажмем кнопку **Выполнить**. Django выведет страницу подтверждения, где присутствует список удаляемых записей и кнопка **Да, я уверен**. Которую мы и нажмем.

Выйти из административного сайта можно, щелкнув расположенную в верхнем левом углу гиперссылку **Выйти**. На очередной странице мы увидим довольно неуклюжее сообщение, говорящее об успешном выходе, и гиперссылку **Войти снова**.

Что дальше?

В этой главе мы создавали и настраивали наш первый проект, выполняли первую синхронизацию с базой данных и генерировали первое приложение Django. Попутно мы познакомились со встроенным административным Web-сайтом, который поможет нам наполнить модели отладочными данными.

Модели, модели, модели — только мы и слышим... Не пора ли заняться ими вплотную и создать, наконец, хоть одну из них?